IPE JC13
OCT 16 2000
PATENT & TRADEMARK OFFICE

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Reissue Appl. No.	:	09/590,584
Filed	:	8 June 2000
In re Patent to	:	Tai A. Ly et al.
Patent No.	:	5,764,951
Issued	:	9 June 1998
Title	:	METHODS FOR AUTOMATICALLY PIPELINING LOOPS

Box REISSUE
Assistant Commissioner for Patents
Washington, D.C. 20231

DECLARATION OF JONATHAN T. KAPLAN AND STATEMENT OF FACTS IN SUPPORT OF FILING ON BEHALF OF NON-SIGNING INVENTOR
Pursuant to 37 CFR 1.47

Jonathan T. Kaplan, being duly warned that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application or any patent resulting therefrom, declares that:

1. I am registered to practice before the Office in patent cases and am attorney of record herein. I am familiar with all aspects of the preparation and filing of the application herein.

2. This is an application for broadening reissue of a previously-issued United States patent. As such it neither discloses nor claims anything not disclosed in the original application for patent.

3. On 31 July 2000, as part of the process of completing formalities associated with the filing of the application herein, I caused to be delivered to each of the four inventors named herein a copy of the specification and a statutory declaration stating that the

THIS PAGE BLANK (USPTO)

inventors were first, original, and joint inventors of the subject matter described and claimed in the application. In each case the specification and declaration were accompanied by a cover letter requesting that the addressee review the application and sign and return the declaration. These letters and their enclosures were sent by Federal Express "priority overnight" service, signature required.

4. By 18 August I received from three of the four inventors, namely Tai A. Ly, Donald B. Macmillen, and Ronald A. Miller, either signed declarations or clear oral indications that the declaration would be signed and returned. By 31 August 2000 I received signed declarations from each of these three inventors.

5. My letter to the fourth inventor, Mr. David W. Knapp, was not answered. This letter and its enclosures were delivered by Federal Express to Mr. Knapp's place of business on 1 August 2000. Acknowledgement of the delivery was made on behalf of Mr. Knapp by a B. Ngyen. A copy of my letter of 31 July 2000 to Mr. Knapp, including copies of the enclosures to that letter and an electronic copy of B. Ngyen's signature acknowledging the letter's receipt, is attached hereto as Exhibit 1.

6. Mr. Knapp's failure to respond to my letter and request of 31 July 2000 was not surprising to me. I had previously received information indicating that Mr. Knapp is now employed by a competitor of the assignee of the patent which is the subject of this reissue application, and that in fact Mr. Knapp's current employer may now be engaged in conduct which would constitute infringement of the reissued patent.

7. Having received no response to my letter of 31 July 2000, I caused to be sent to Mr. Knapp a second letter on 18 August 2000. This second letter referred to and reiterated the contents of the first and, like the first, enclosed a copy of the specification herein and a statutory declaration for Mr. Knapp's signature. This letter and its enclosures were again delivered by Federal Express to Mr. Knapp's business address. Delivery was completed on 21 August. Acknowledgement of the delivery was made on Mr. Knapp's

THIS PAGE BLANK (USPTO)

behalf by an E. Castor. A copy of my letter of 18 August 2000 to Mr. Knapp, including copies of the enclosures to that letter and an electronic copy of E. Castor's signature acknowledging the letter's receipt, is attached hereto as Exhibit 2.

8. On or about 1 September 2000 I received a letter from Mr. Knapp in response to my letter of 31 July 2000. This letter bears Mr. Knapp's signature and states that Mr. Knapp will neither sign the declaration nor otherwise cooperate in prosecuting the application. A copy of Mr. Knapp's letter is attached hereto as Exhibit 3.

9. The facts set forth in this Declaration are true, and all statements made of my own knowledge are true, and all statements made on information and belief are believed to be true.

Dated: Oct. 11, 2000

By: Jonathan T. Kaplan
Jonathan T. Kaplan
Registration No. 38,935
Attorney for Assignee

THIS PAGE BLANK (USPTO)

EXHIBIT 1

THIS PAGE BLANK (USPTO)

EXHIBIT 1

TO

DECLARATION OF JONATHAN T. KAPLAN AND STATEMENT OF FACTS IN SUPPORT OF FILING ON BEHALF OF NON-SIGNING INVENTOR

Pursuant to 37 CFR 1.47

THIS PAGE BLANK (USPTO)

FedEx Express
Customer Support Trace
3875 Airways Boulevard
Module H, 4th Floor
Memphis, TN 38116

U.S. Mail: PO Box 727
Memphis, TN 38194-4643

Telephone: 901-369-3600

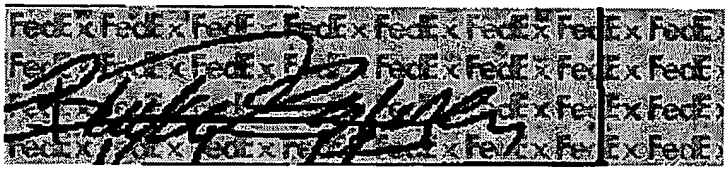
8/4/2000

Dear Customer:

Here is the proof of delivery for the shipment with tracking number **820573308506**. Our records reflect the following information.

Delivery Information:

Signed For By: B.NGYEN



Delivery Location: 2107 N 1ST ST 350
Delivery Date: August 1, 2000
Delivery Time: 1019

Shipping Information:

Tracking No: 820573308506

Ship Date: July 31, 2000

R cipient:
MR DAVID W KNAPP
GET 2 CHIP COM
2107 N FIRST ST 350
SAN JOSE, CA 95131
US

Shipper:
KAPLAN JONATHAN T
BROWN RAYSMAN ET AL
120 W 45TH ST FL 21
NEW YORK, NY 100364041

Shipment Reference Information: 4000/10

Thank you for choosing FedEx Express. We look forward to working with you in the future.

FedEx Worldwide Customer Service
1-800-Go-FedEx®
Reference No.: R2000080400019203123

THIS PAGE BLANK (USPTO)

THIS PAGE BLANK (USPTO)

Track Shipments
Detailed Results

Quick Help

View detailed results for: Previous 820573308506 Next

Tracking Number	820573308506
Reference Number	
Ship Date	07/31/2000
Delivered To	Recept/Frnt desk
Delivery Location	SAN JOSE CA
Delivery Date/Time	08/01/2000 10:19
Signed For By	B.NGYEN
Service Type	Priority Letter

Tracking Options
- Obtain a Signature Proof of Delivery
- Email these tracking results to one or more recipients
- Return to Summary Results
- Track More Shipments

Scan Activity	Date/Time	Comments
Delivered SAN JOSE CA	08/01/2000 10:19	
Placed on Van SAN JOSE CA	08/01/2000 08:20	
Left FedEx Sort Facility OAKLAND CA	08/01/2000 05:32	
Arrived at Sort Facility OAKLAND CA	08/01/2000 05:17	
Arrived at FedEx Destination Location SAN JOSE CA	08/01/2000 07:50	
Arrived at Sort Facility MEMPHIS TN	08/01/2000 01:24	
Left FedEx Sort Facility MEMPHIS TN	08/01/2000 04:49	
Left FedEx Sort Facility MEMPHIS TN	08/01/2000 02:10	
Left FedEx Sort Facility NEWARK NJ	08/01/2000 00:25	
Left FedEx Sort Facility NEWARK NJ	07/31/2000 22:19	
Left FedEx Origin Location NEW YORK NY	07/31/2000 22:45	
Left FedEx Origin Location NEW YORK NY	07/31/2000 19:35	
Left FedEx Origin Location NEW YORK NY	07/31/2000 21:54	

Email Your Detailed Tracking Results

Enter your email (optional), up to three email addresses as recipients, add your message, and click on **Send Email**.

From

To

To

To

Add a message to this email.

Send Email

THIS PAGE BLANK (USPTO)

BROWN RAYSMAN MILLSTEIN FELDER & STEINER LLP

120 WEST FORTY-FIFTH STREET • NEW YORK, NY 10036 • TELEPHONE: 212 944 1515 • FACSIMILE: 212 840 2429

JONATHAN T. KAPLAN
PARTNER
212 827 9478
E-MAIL: jkaplan@brownraysman.com
WEB SITE: www.brownraysman.com

31 July 2000

VIA FEDERAL EXPRESS
(SIGNATURE REQUIRED ON RECIEPT)

Mr. David W. Knapp
Chief Technical Officer
Get2Chip.com, Inc.
2107 North First Street, Suite 350
San Jose, California 95131

Re: Reissue of U.S. Patent 5,764,951 to Ly et al.
METHODS FOR AUTOMATICALLY PIPELINING LOOPS
Our File Ref. 4000/10

Dear Mr. Knapp:

This law firm represents your former employer Synopsys, Inc. (hereinafter "Synopsys"). Synopsys is currently seeking reissuance of the above-identified patent in the United States Patent and Trademark Office. Enclosed please find a Declaration of Inventor which we will be filing in the Reissue Patent Application. Also enclosed is a copy of the Reissue Application, which adds new claims 23-34 to those which already issued in U.S. Patent 5,764,951. Please review the Declaration and the new claims carefully.

Assuming you understand and agree with the Declaration, Synopsys requests that you do the following. Please complete the Declaration by entering, in the boxes at the end of the Declaration, your country of citizenship and the address of your current home residence. Once you have completed the Declaration, please sign the Declaration in the indicated box at the end of the Declaration. Return the Declaration to us by means of the enclosed self-addressed stamped envelope.

IT IS VERY IMPORTANT THAT YOU RETURN THE DECLARATION TO US BY SEPTEMBER 1, 2000. IF YOU HAVE ANY QUESTIONS ABOUT THE DECLARATION, IT IS EXTREMELY IMPORTANT THAT YOU CONTACT US AS SOON AS POSSIBLE SO THAT WE MAY ANSWER YOUR QUESTIONS.

As I am sure you will recall, you have previously assigned all rights in the invention to Synopsys as your former employer.

THIS PAGE BLANK (USPTO)

Your cooperation in this matter is greatly appreciated. On behalf of Synopsys, Inc., I thank you most sincerely for your help.

Very truly yours,

Jonathan T. Kaplan

Jonathan T. Kaplan

JTK/mjm
Enclosures Declaration Of Inventor (w/ self-addressed stamped envelope)
Reissue Application

THIS PAGE BLANK (USPTO)

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Reissue Appl. No.	:	09/590,584
Filed	:	June 8, 2000
In re Patent to	:	Tai A. Ly et al.
Patent No.	:	5,764,951
Issued	:	June 9, 1998
Title	:	METHODS FOR AUTOMATICALLY PIPELINING LOOPS

Box REISSUE
Assistant Commissioner for Patents
Washington, D.C. 20231

DECLARATION OF INVENTOR DAVID W. KNAPP IN APPLICATION FOR BROADENING REISSUE OF PATENT
Pursuant to 37 CFR 1.63 and 1.175

This declaration is made in application for broadening reissue of the above-identified patent.

As a below named inventor, I hereby declare that:

My residence, post office address and citizenship are as stated below next to my name.

I believe I am an original, first and joint inventor of the subject matter which is claimed and for which a patent is sought on the invention entitled

METHODS FOR AUTOMATICALLY PIPELINING LOOPS

the specification of which: *(check one)*
☐ is attached hereto; or
☒ was filed on June 8, 2000 as U.S. Reissue Application Serial No. 09/590,584.

I hereby state that I have reviewed and understand the contents of the above identified specification, including the claims, as amended by any amendment referred to above.

I acknowledge the duty to disclose information which is material to patentability as defined in Title 37, Code of Federal Regulations, §1.56.

THIS PAGE BLANK (USPTO)

I believe the original patent to be wholly or partly inoperative or invalid, for the reasons described below:

☐ by reason of a defective specification or drawing.
☒ by reason of the patentee claiming more or less than he had the right to claim in the patent.
☐ by reason of other errors.

At least one error upon which this application for reissue is based is described as follows:

The limitation of claim 1 to methods comprising steps of parsing text descriptions including loops with delayed signal assignments having delay values and setting latencies of pipelines equal to said delay values is more limiting than necessary, and resulted in the patentee claiming less than he had a right to claim.

The limitation of claim 18 to systems comprising logic for parsing text descriptions including loops with delayed signal assignments having delay values and setting latencies of pipelines equal to said delay values is more limiting than necessary, and resulted in the patentee claiming less than he had a right to claim.

The limitation of claim 21 to computer program products comprising computer readable program code devices configured to cause a computer effect parsing of text descriptions including loops with delayed signal assignments having delay values and setting of latencies of pipelines equal to said delay values is more limiting than necessary, and resulted in the patentee claiming less than he had a right to claim.

All errors being corrected in this reissue application arose without any deceptive intention on the part of the applicant.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Full Name of First Joint Inventor	David W. Knapp		
Inventor's Signature		Date	
Residence		Citizenship	
Post Office Address			

THIS PAGE BLANK (USPTO)

METHODS FOR AUTOMATICALLY PIPELINING LOOPS

Related Applications

This application is related to U.S. patent application Ser. No. 08/440,101 entitled "Behavioral Synthesis Links to Logic synthesis" with inventors Ronald A. Miller, Donald B. MacMillen, Tai A. Ly and David W. Knapp filed on May 12, 1995, which is hereby incorporated by reference.

Background

Field of the Invention

This invention relates to the field of computer aided design for digital circuits, particularly to automatically pipelining loops in a behavioral synthesis system.

Statement of the Related Art

Behavioral Synthesis

Behavioral vs. Register Transfer Level Design

Many of today's integrated circuits are described using a Hardware Description Language (HDL). Two common HDL's are VHDL and Verilog. VHDL is described in the IEEE Standard VHDL Language Reference Manual available from the Institute of Electrical and Electronic Engineers in Piscataway, New Jersey which is hereby incorporated by reference. Verilog is described in The Verilog Hardware Description Language by Donald E. Thomas and Philip Moorby, Kluwer Academic Publishers, 1991 which is hereby incorporated by reference.

As integrated circuits become increasingly complex, hardware designers are increasingly using *synthesis* software to transform HDL descriptions of digital circuits into mapped logic. The designer writes a description of a digital circuit in VHDL, Verilog, or another HDL, and uses synthesis software to create a digital circuit from the description. Using synthesis software typically shortens the amount of time required to create a digital circuit from a design specification, and allows a designer to create more complex designs than is possible manually.

Many of today's complex designs are expressed as software descriptions and

simulated to verify their correctness. These designs are later translated from software into hardware, in the form of Integrated Circuits (ICs), Application Specific Integrated Circuits (ASICs), or Field Programmable Gate Arrays (FPGAs), for implementation in the final product. This design description methodology is called *algorithmic-level* design.

Instead of beginning design at the Register Transfer Level (RTL), behavioral synthesis begins at the algorithmic (behavioral) level. RTL level design is described in Computer Structures: Reading and Examples by C. Gorden Bell and Allen Newell, McGraw-Hill 1971. A behavioral hardware description language (HDL) specification contains instructions, operations, variables, and arrays similar to the original software algorithm.

The target architecture of behavioral synthesis is a general computing model that contains datapath, memory, and control elements. Conventional design techniques currently use a manual RTL design methodology to build a datapath. A datapath is a sequence of logic consisting of registers, higher order functional units (such as adders and multipliers), and multiplexers. The datapath in a digital circuit uses the circuit's inputs to compute output results. Registers are 1-bit memory elements which hold their value through each clock cycle.

Conventional design techniques also build a controller at the RTL to sequence and control the actions of the datapath, memory, and Input/Output (I/0). Frequently, such controllers are implemented using a Finite State Machine (FSM). Finite state machines are described in Switching and Finite Automata Theory by Zvi Kohavi, Computer Science Press, 1978 which is hereby incorporated by reference. Controllers may also determine actions such as which branch of a conditional statement is executed.

Behavioral synthesis builds this architecture by using automated methods of scheduling, allocation, register sharing, memory and control inferencing--all of which are performed manually in an RTL methodology. The designer is freed from having to specify the exact architecture of a design and can automatically explore many implementations to find the optimal architecture.

Components of Behavioral Synthesis

The High-Level Synthesis of Digital Systems by Michael McFarland, Alice Parker, and Raul Camposano, in Proceedings of the IEEE, February 1990, which is hereby incorporated by reference, provides an excellent overview of High Level Synthesis, as Behavioral Synthesis is often called.

Three components of a behavioral synthesis system are Scheduling, Allocation, and Resource Sharing.

Scheduling determines in which clock cycle each operation executes. Scheduling extracts the control and data flow operations of a design specification and assigns these operations to cycles. A state machine controller is synthesized to sequence the operations and execute them in their assigned cycle. The typical goal of this process is to assign operations to cycles so as to be able to implement the design with the fewest resources (registers, multiplexers, and operations) while at the same time minimizing the number of clock cycles (latency).

Allocation is a behavioral synthesis task that maps the operations and data of a behavioral HDL specification into the datapath, which contains memories, registers, functional units such as adders and multiplexers, and gates. Allocation determines which type of operation to use for each operator. For instance, if an operator performs addition, a ripple carry, a carry-lookahead, or some other type of adder can be used.

Resource Sharing attempts to share hardware resources between operators in a design. For example, consider two additions which occur in mutually exclusive conditional branches. Such additions will never be performed at the same time. Thus, they can be performed on the same piece of hardware. Resource sharing attempts to minimize the amount of hardware used by sharing hardware as much as possible.

Scheduling Modes

There are several modes for automatically scheduling operations into control steps. Briefly, these modes are cycle-fixed, superstate-fixed, and free-floating mode. In cycle-fixed mode, all I/O operations are constrained to occur in the same cycle in the original HDL descriptions and in the synthesized design. In cycle-fixed mode, the cycle

level behavior of the synthesized circuit must match the cycle level simulation behavior of the source HDL.

The other scheduling modes allow behavioral synthesis a greater degree of freedom in assigning states in a schedule. Scheduling modes are discussed further in Behavioral Synthesis Methodology for HDL-Based Specification and Validation by D. Knapp, T. Ly, D. MacMillen and R. Miller in Proceedings of the 31st DAC, June 1995, which is included as Appendix B and is hereby incorporated by reference. They are also discussed in Behavioral Compiler User Guide Version 3.2a available from Synopsys, Inc. in Mountain View, Calif., which is hereby incorporated by reference.

Loop Pipelining

In behavioral HDL, a loop repeatedly executes the operations in the loop body until an exit condition becomes true. Loop iterations are usually sequential; operations in the first iteration are executed, operations in the next iteration are executed, and so on, as shown in Figure 1. The throughput, that is the amount of data processed per unit time, of the function implemented by the loop body is limited by the critical path in the loop body.

In some loops, data required by an operation in the next loop iteration is available prior to completion of the current loop. Under these conditions, the designer can *pipeline* the loop--parallelizing execution of iterations to increase throughput beyond critical path limitations of the loop body. This process of loop pipelining schedules consecutive loop iterations to partially overlap in time; a new loop iteration is initiated before the current iteration has finished.

Figure 2 shows an example of loop pipelining where the data required by operation A in iteration two is available after operation C in the first loop iteration.

The two timing-related aspects of a loop that affect throughput are:

Initiation interval: The number of clock cycles between the start of two consecutive loop iterations.

Latency: The number of clock cycles required to execute all operations in a single loop iteration.

For sequential loops that are not pipelined, the initiation interval and latency of a loop are the same. For a pipelined loop, the initiation interval is smaller than the latency.

The primary reason for using loop pipelining is to increase the throughput of the design; the trade-off is that the design area usually increases.

Many designs have separate specifications on throughput and input-to-output delay. The throughput specification constrains the initiation interval. The input-to-output delay specification constrains the loop latency. Loop pipelining enables a flexible relationship between the initiation interval and latency of a loop.

An example of a candidate for loop pipelining is a design that processes a data stream. This type of design often has tight throughput requirements based on the rate of the data streams and loose input-to-output delay constraints.

Loop Carry Dependencies

Loop Carry Dependencies (LCDs) are data values produced in one iteration of a loop and consumed by operations in subsequent iterations.

In loop pipelining, loop iterations that are producers and consumers of LCDs can happen at the same time. To preserve data dependencies, the operations in a loop must be scheduled so that LCD values are available in time for the iteration in which they will be consumed. Two schedules for a LCD are shown in Figure 3.

The example of Figure 3(a) violates the LCD. Operation 410 is scheduled so that its output is not ready in time for operation 420 to use it in the next iteration of the loop. The example of Figure 3(b) is scheduled correctly. In this case, operation 410 is scheduled so that its output is ready in time for operation 420 to execute in the next iteration of the loop.

Memory and I/O Accesses

Loop Pipelining must preserve the original ordering of all reads and writes to the same memory, signal, or port. In addition, the ordering reads and writes in one iteration of the loop may not "cross," or occur after, reads and writes in subsequent iterations of the loop. Specifically, all reads and writes to the same memory, and all writes to the

same signal or port in one iteration of the loop must occur before any reads or writes to the same memory, signal or port in a subsequent iteration of the loop. All reads of the same signal or port must occur simultaneously to or before any read of the same signal or port in a subsequent iteration of the loop.

For example, Figure 4 shows two schedules for a loop that has two reads of signal x. In FIGURE 4(a), read 510 and read 520 are improperly scheduled. Read 520 occurs after read 510 occurs in the next iteration of the loop. In FIGURE 4(b), read 510 and read 520 are properly scheduled. In this schedule, read 520 occurs after read 510 in the next iteration of the loop.

A Brief Description of the Drawings

The accompanying drawings, which are incorporated in and constitute a part of this specification, illustrate several embodiments of the invention and, together with the description, serve to explain the principles of the invention.

Figure 1 shows an example of sequential loop processing.

Figure 2 shows an example of pipelined loop processing including the loop latency and initiation interval.

Figure 3 shows an example of a loop carry dependency.

Figure 4 shows an example of memory and I/O access restrictions in pipelined loops.

Figure 5 is a block diagram showing a computer system.

Figure 6 is a flowchart which shows steps in a circuit synthesis process.

Figure 7 is a flowchart which shows steps for scheduling preprocessing.

Figure 8 is a flowchart which shows steps for inserting constraints into a constraint graph.

Figure 9 is a flowchart which shows steps for scheduling templates.

Figure 10 is a flowchart which shows steps for creating a constraint using templates.

Figure 11 shows HDL source code which contains a loop with a producer and a consumer.

Figure 12 shows a circuit before scheduling which is created from loop 3030 of Figure 11.

Figure 13 shows a constraint created for a producer and consumer in loop 3030.

Figure 14 shows a circuit which is created after scheduling loop 3030 using an initiation interval of 2 and a latency of 4.

Figure 15 shows Verilog HDL source code which contains a loop with I/O dependencies.

Figure 16 shows a circuit before scheduling which is created from loop 1530 of Figure 15.

Figure 17 shows a constraint created for two reads in loop 1530.

Figure 18 shows a circuit which is created after scheduling loop 1530 using an initiation interval of 2 and a latency of 4.

Figure 19 (a) and Figure 19 (b) are examples of HDL source code including a delay clause.

Figure 20 is a flowchart showing steps performed during translation from the source code of Figure 19 (a) and Figure 19 (b) to a circuit design that incorporates a delay specified by the delay clause.

Figure 21 is a representation of a data flow graph generated from the source code of Figure 19 (a) and (b) in accordance with the steps of Figure 20.

Figure 22 is a representation of a control flow graph generated from the source code of Figure 19 (a) and (b) and the data flow graph of Figure 21.

Figure 23 is a flow chart showing steps performed to generate a control data flow graph from the control flow graph and data flow graph of Figure 21 and Figure 22.

Figure 24 is a representation of a control data flow graph generated by the steps of Figure 23.

Figure 25 is a diagram showing an example of loop tiling with and without the delay in the HDL.

Figure 26 is a diagram showing the effect of the delay clause on pipelining.

Figure 27 shows the operations of Figure 12 scheduled into control steps.

Figure 28 shows the read operations of Figure 16 scheduled into control steps.

Detailed Description of the Invention

The present invention is a method and apparatus for synthesizing a circuit which implements a pipelined loop from a Hardware Description Language (HDL) description. The following description is presented to enable any person skilled in the art to make and use the invention, and is provided in the context of a particular application and its requirements. Various modifications to the preferred embodiment will be readily apparent to those skilled in the art, and the generic principles defined herein may be applied to other embodiments and applications without departing from the spirit and scope of the invention. Thus, the present invention is not intended to be limited to the embodiment shown, but is to be accorded the widest scope consistent with the principles and features disclosed herein.

1.0 Computer System Description

Figure 5 illustrates a computer system 100 in accordance with a preferred embodiment of the present invention. The computer system 100 includes a bus 101, or other communications hardware and software, for communicating information, and a processor 109, coupled with the bus 101, is for processing information. The processor 109 can be a single processor or a number of individual processors that can work together. The computer system 100 further includes a memory 104. The memory 104 can be random access memory (RAM), or some other dynamic storage device. The memory 104 is coupled to the bus 101 and is for storing information and instructions to be executed by the processor 109. The memory 104 also may be used for storing temporary variables or other intermediate information during the execution of instructions by the processor 109. The computer system 100 also includes a ROM 106 (read only memory), and/or some other static storage device, coupled to the bus 101. The ROM 106 is for storing static information such as instructions or data.

The computer system 100 can optionally include a data storage device 107, such as a magnetic disk, a digital tape system, or an optical disk and a corresponding disk drive. The data storage device 107 can be coupled to the bus 101.

The computer system 100 can also include a display device 121 for displaying

information to a user. The display device 121 can be coupled to the bus 101. The display device 121 can include a frame buffer, specialized graphics rendering devices, a cathode ray tube (CRT), and/or a flat panel display. The bus 101 can include a separate bus for use by the display device 121 alone.

An input device 122, including alphanumeric and other keys, is typically coupled to the bus 101 for communicating information, such as command selections, to the processor 109 from a user. Another type of user input device is a cursor control 123, such as a mouse, a trackball, a pen, a touch screen, a touch pad, a digital tablet, or cursor direction keys, for communicating direction information to the processor 109, and for controlling the cursor's movement on the display device 121. The cursor control 123 typically has two degrees of freedom, a first axis (e.g., x) and a second axis (e.g., y), which allows the cursor control 123 to specify positions in a plane. However, the computer system 100 is not limited to input devices with only two degrees of freedom.

Another device which may be optionally coupled to the bus 101 is a hard copy device 124 which may be used for printing instructions, data, or other information, on a medium such as paper, film, slides, or other types of media.

A sound recording and/or playback device 125 can optionally be coupled to the bus 101. For example, the sound recording and/or playback device 125 can include an audio digitizer coupled to a microphone for recording sounds. Further, the sound recording and/or playback device 125 may include speakers which are coupled to digital to analog (D/A) converter and an amplifier for playing back sounds.

A video input/output device 126 can optionally be coupled to the bus 101. The video input/output device 126 can be used to digitize video images from, for example, a television signal, a video cassette recorder, and/or a video camera. The video input/output device 126 can include a scanner for scanning printed images. The video input/out-put device 126 can generate a video signal for, for example, display by a television.

Also, the computer system 100 can be part of a computer network (for example, a LAN) using an optional network connector 127, being coupled to the bus 101. In one

embodiment of the invention, an entire network can then also be considered to be part of the computer system 100.

An optional device 128 can optionally be coupled to the bus 101. The optional device 128 can include, for example, a PCMCIA card and a PCMCIA adapter. The optional device 128 can further include an optional device such as modem or a wireless network connection.

2.0 Definitions

A digital circuit is an interconnected collection of parts. Parts may also be called cells. The digital circuit receives signals from external sources at points called primary inputs. The digital circuit produces signals for external destinations at points called primary outputs. Primary inputs and primary outputs are also called ports. Each part receives input signals and computes output signals. Each part has one or more pins for receiving input signals and producing output signals. In general, pins have a direction. Most pins are either input pins, which are called loads, or output pins, which are called drivers. Some pins may be bidirectional pins, which can be both drivers and loads.

Two or more pins from one or more parts or primary inputs or primary outputs are connected together with a net. Each net establishes an electrical connection among the connected pins, and allows the parts to interact electrically with each other. Pins are also connected to primary inputs and primary outputs with nets. For the sake of simplicity, parts may be said to be "connected" to nets, but it is actually pins on the parts which are connected to the nets.

A Circuit Element is any component of a circuit. Ports, pins, nets, and cells are all circuit elements. Any circuit element which is an input to another circuit element is said to drive that circuit element. Any circuit element which is an output of another circuit element is said to load that circuit element. For example, drivers drive a signal onto a net; loads load nets with capacitance.

A digital circuit design can be stored in memory of a computer system using data structures which represent the various components of the circuit. The data structures have the same name as the physical components. In this document, parts,

cells, nets, pins, and other digital circuit components refer to the software representation of the physical digital circuit component.

A digital circuit can be specified hierarchically. Some or all of the parts in the digital circuit may themselves be digital circuits composed of more interconnected parts. When a high level part is specified as a digital circuit composed of other, lower level parts, the pins of the high level part become the primary inputs and primary outputs for the digital circuit comprising the lower level parts. When a high level part is composed of lower level parts, it is called a level of hierarchy.

Following are additional definitions of terms which are used in this document.

An *HDL* is a Hardware Description Language. HDL's are used to describe designs for digital circuits.

A *Translated Circuit, Generic Technology Circuit, or GTech Circuit* is a software representation of a digital circuit which does not include references to a specific technology, but rather refers to cells that implement generic logic such as "and", "or", and "not". This software representation is stored in memory 104 of computer system 100.

A *Mapped Circuit* is a software representation of a digital circuit which is built from parts available in a technology library which is provided by a silicon vendor. This software representation is stored in memory 104 of computer system 100. A mapped circuit can be timed using a conventional timing verifier such as DesignTime, available from Synopsys, Inc. in Mountain View, Calif. After it is built, a netlist representation of a mapped circuit can be sent to a silicon vendor for layout and fabrication. For instance, the mapped circuit can be written out using LSI netlist format and sent to LSI Logic in Milpitas, Calif. The process of creating a mapped circuit from a generic technology circuit is called mapping. Because a circuit must be mapped before it can be timed, mapped circuits are also used internally by synthesis tools.

The *Fanout* of a circuit element includes any circuit elements which are driven by that circuit element. The *transitive fanout* of a circuit element includes all of the circuit elements in the circuit which are driven, either directly or indirectly, by that

circuit element. Thus, the transitive fanout of a circuit element includes the fanout of that circuit element, as well as the fanout of each of the circuit elements in the original fanin, and so on.

The *Fanin* of a circuit element includes any circuit elements which drive that circuit element. The *transitive fanin* of a circuit element includes all of the circuit elements in the circuit which drive, either directly or indirectly, that circuit element. Thus, the transitive fanin of a circuit element includes the fanin of that circuit element, as well as the fanin of each of the circuit elements in the original fanin, and so on.

An *Operator* is a function, such as addition. Such functions are used in HDL source code. For example, the plus in "c=a+b;" is an operator.

An *Operation* is a software representation of a hardware functional unit which performs a function such as addition. For example, a software representation of an adder is an operation.

A *Clock Cycle* is a period of time, for example 10ns, between pulses of a clocking element in a digital circuit. The clocking element is used to synchronize the digital circuit.

3.0 Scheduling

Scheduling is a well defined problem which has been studied extensively. An overview of the scheduling problem is available in The High-Level Synthesis of Digital Systems by Michael McFarland, Alice Parker, and Raul Camposano, in Proceedings of the IEEE, February 1990, which is hereby incorporated by reference.

The input to a scheduler is typically a set of hardware operations, a set of constraints between the hardware operations, a clock period, and a set of control steps into which the hardware operations must be mapped. The output is a schedule where each hardware operation is mapped to a control step.

Schedulers typically use a number of graphs. For instance, the constraints for a scheduler are often represented using a graph. Nodes in the graph typically represent events to be scheduled, such as operations, and edges in the graph represent constraints between the events. The scheduler checks the constraint graph to ensure that all of the

constraints are met before placing an event into a particular control step. Schedulers also use control graphs, data flow graphs, and combination control data flow graphs (CDFG's). Control graphs represent the flow of control in a circuit. Data flow graphs represent the flow of data in a circuit; that it the flow of data from the inputs to the outputs of the circuit. Control data flow graphs combine both control flow and data flow information into a single graph. All of these types of graphs are described in High-Level Synthesis (subtitled Introduction to Chip and System Design) by Daniel Gajski, Nikil Dutt, Allen C-H Wu, and Steve Y-L Lin, Kluwer Academic Publishers, 1992 which is hereby incorporated by reference and will subsequently be referred to as High-Level Synthesis by Gajski et al.

An additional technique used for scheduling circuits involves "templates". Templates are described in Scheduling using Behavioral Templates by Tai Ly, David Knapp, Ron Miller, and Don MacMillen in Proceedings of the 31st DAC, June 1995, which is included as Appendix A and is hereby incorporated by reference. Simply speaking, templates are data structures which specify scheduling constraints among CDFG nodes. Templates "lock" the control step relationship between 2 or more CDFG nodes. Figure 13 shows an example of two templates, template 1250 and template 1280. Each template contains one or more nodes, some of which may represent operations. For example, adder node 2020 represents adder 3120 of Figure 12.

3.1 Overview of Synthesis with Scheduling

Figure 6 is a flowchart showing how scheduling steps fit into the overall synthesis strategy. This flowchart shows how a mapped circuit is created from a source HDL description. The input to synthesis is an HDL description of a digital circuit. Such a description may be written in VHDL, Verilog, or some other HDL.

An HDL description is translated in step 810 to generic logic. A conventional HDL translator 1310 such as VHDL Compiler version 3.2b from Synopsys, Inc. in Mountain View, Calif. preferably is used.

Step 820 performs scheduling preprocessing steps. These steps are shown in Figure 7 and Figure 8.

Step 830 schedules the operations in the circuit. A method for scheduling the operations in the circuit is shown in Figure 9.

Step 840 netlists the scheduled circuit. Netlisting creates a GTech circuit from the scheduled CDFG. The CDFG representation of the circuit in memory is transformed into a GTech representation of the circuit in memory.

In step 850, the resulting GTech circuit is optimized using conventional logic synthesis such as Design Compiler version 3.2 b by Synopsys, Inc. in Mountain View, Calif. The output of logic optimization is a mapped circuit description which can be sent to a silicon vendor for fabrication. For example, a description of the mapped circuit can be output using LSI Netlist format and sent to LSI Logic in Milpitas, Calif for fabrication.

3.2 Scheduling Preprocessing

Figure 7 is a flowchart which shows steps for scheduling preprocessing. The input to the method is an annotated GTech circuit. Annotations on the circuit include delayed signal assignment information. The use of delayed signal assignments will be discussed in a later section.

Step 910 extracts a control graph from the annotated GTech using conventional techniques. In addition, information concerning delayed signal assignments is extracted as described below.

Step 920 extracts a Control Data Flow Graph (CDFG) from the control graph created in step 910 and the data flow graph represented by the GTech circuit. This is also done using conventional techniques.

Step 930 creates initial templates for the operations in the CDFG as described in Scheduling Using Behavioral Templates in Appendix A. These initial templates form the initial constraint graph.

Step 940 inserts constraints in the constraint graph. Some types of constraints are discussed in Scheduling Using Behavioral Templates in Appendix A. Other types of constraints are a part of the present invention and will be discussed in subsequent sections.

3.3 Inserting Constraints

Step 940 of Figure 7 is implemented by Figure 8 which is a flowchart which shows steps for inserting constraints into a constraint graph which uses templates. The input to the process is a CDFG and a constraint graph.

Step 1110 identifies Loop Carry Dependency (LCD) producer consumer pairs. LCD's are identified by tracing the CDFG using conventional techniques. LCD's are discussed below in connection with Figure 11, Figure 12, Figure 13, Figure 14, and Figure 27.

Step 1120 constrains the LCD's. Constraining LCD's involves adding constraints to the constraint graph so that producer and consumer operations are scheduled so that the consumer consumes a value produced by the producer before it is overwritten in a subsequent iteration of the loop. A method and apparatus for constraining LCD's will be discussed in a later section.

Step 1130 identifies memory and I/O access dependencies in loops which will be scheduled using pipelines. I/O accesses include reads and writes to memories, signals, and ports. Reads and writes in one iteration of the loop may not "cross," or occur after, reads and writes in subsequent iterations of the loop. Specifically, all reads and writes to the same memory, and all reads and writes to the same signal or port in one iteration of the loop must occur before any reads or writes to the same memory, signal or port in a subsequent iteration of the loop. The one exception to this rule is that reads of the same signal or port may occur simultaneously to a read of the same signal or port in a subsequent iteration of the loop. This step finds the first and last accesses for each memory, signal, or port by tracing through the CDFG using conventional techniques. Memory and I/O accesses are discussed below in connection with Figure 15, Figure 16, Figure 17, Figure 18, and Figure 28.

Step 1120 constrains the memory and I/O accesses in pipelined loops. Constraining memory and I/O accesses involves adding constraints to the constraint graph so that first and last accesses are scheduled so that the last access occurs before the first access in a subsequent iteration of the loop. A method and apparatus for

constraining memory and I/O accesses will be discussed in a later section.

Step 1130 inserts other types of constraints into the constraint graph. Such constraints are discussed in Scheduling Using Behavioral Templates in Appendix A. An example of another type of constraint is a dataflow constraint, which ensures that data values are produced before they are consumed by subsequent operations.

3.4 Scheduling Templates

Figure 9 is a flowchart which shows steps of scheduling (step 830 of Figure 6) using templates. The input to the process is the CDFG and the constraint graph created by the steps of Figure 8. It is possible to schedule templates using many different scheduling techniques. A number of scheduling techniques are described in High-Level Synthesis by Gajski et al, particularly in Chapter 7. This figure shows a general method, which is provided as an example.

Step 1010 creates the As Soon As Possible (ASAP) and As Late As Possible (ALAP) schedules for each template while satisfying the constraints represented in the constraint graph. The ASAP schedule places each template into the earliest possible control step (c-step). The ALAP schedule places each template into the latest possible control step. Together, the earliest and latest control steps define a range into which each template may be scheduled. A method for determining the ASAP and ALAP schedules for templates is described in Scheduling Using Behavioral Templates in Appendix A.

Loop 1020 loops until a "good" schedule is found. A "good" schedule is one which fulfills the constraints specified in the constraint graph and optimizes for a specific goal specified by a human designer, such as fewest number of control steps. Different scheduling techniques use different criteria for deciding when to stop trying to improve the schedule. For example, one technique might stop when the constraints are all met, or when a certain amount of CPU time has been spent, whichever comes last.

Step 1030 picks a template in the constraint graph to schedule. Different techniques use different criteria for deciding what to schedule next. Generally, template scheduling techniques use criteria based upon the operations in a template. For instance,

a list scheduling technique which uses priorities will assign a priority to a template based on the priorities of the operations within the template. (List scheduling is described in High-Level Synthesis by Gajski et al in Chapter 7).

Step 1040 schedules the chosen template in the control step chosen by the scheduling technique being used. Templates are scheduled by placing the first operation within the template into the chosen control step and the remaining operations within the template into subsequent control steps as defined by the template.

Arrow 1050 indicates that loop 1020 iterates until a "good" schedule is found.

4.0 Method for Creating Constraints

This section describes a general technique for constraining the relationship between two nodes in a constraint graph. Such constraints are added in step 940 of Figure 7. The section then describes examples of using this technique to constrain loop carry dependencies and I/O dependencies.

4.1 Placeholder Node Method

Figure 10 shows a general method for creating a scheduling constraint between two nodes in a constraint graph. Such constraints are created in step 1120 and step 1140 of Figure 8 to constraint LCD's and memory and I/O accesses. This section shows a general method and discusses specific examples. The first example constrains an LCD; the second example constrains a pair of signal reads. The input to the process of Figure 10 is a constraint graph, two templates in the graph, Event 1 and Event 2, an integer n, and a number of cycles c. "n" is the number of cycles within which Event 2 must be scheduled after Event 1. "c" is either 0 or 1. "c" has value 0 when Event 2 must be schedule before n cycles after Event 1, and value 0 when Event 2 may be scheduled exactly n cycles after Event 1.

Step 610 adds a placeholder node H to the template for Event 1 in the constraint graph. A placeholder rode is a node in the constraint graph which is only used to create constraints. The placeholder node does not represent any portion of the final circuit. Placeholder node H is inserted into the Event 1's template such that it is locked n cycles after Event 1.

Step 620 adds a constraint in the constraint graph from Event 2 to placeholder node H which constrains Event 2 to occur c cycles before placeholder node H, where c is 0 or 1. The value of c depends on the constraint being added and will be discussed in subsequent sections.

4.2 Using Placeholder Nodes for Loop Carry Dependencies

The following section provides an example of constraining loop carry dependencies using placeholder nodes. Such constraints are created in step 1120 of Figure 8. A loop carry dependency is a data value which is produced in one iteration of a loop and consumed by operations in subsequent iterations of the loop. To use the placeholder node method to schedule loop carry dependencies, Event 1 is set to be the operation which consumes the data. Event 2 is set to be the operation which produces that data. Event 2 must be scheduled so that the correct data values are driving it when it feeds its outputs to Event 1. If the consumer (Event 1) consumes the data one iteration after the producer (Event 2) creates it, then n is set to be the initiation interval of the loop. If the consumer consumes the data k iterations after it is created by the producer, then n is set to be k * initiation interval. For LCD's, "c" has value "1" because the producer must be scheduled before the consumer in the subsequent iteration of the loop.

Figure 11 shows an example of Verilog source code for a loop 3030 with a loop carry dependency between addition 3020 and subtraction 3010. The output of addition 3020, p, drives the input of subtraction 3010 on the next iteration of the loop. "p" is a Loop Carry Dependency. In this example, a human designer has specified that loop 3030 will be scheduled using an initiation interval of 2 and a latency of 4. Although this loop would not usually be pipelined because pipelining does not increase its throughput, this simple example is used for the sake of clarity.

Figure 12 shows a GTech circuit representation 2000 which is created for loop 3030 in Figure 11. The GTech circuit representation is stored in memory 104. GTech circuit 2000 is output from step 810 of Figure 6. Addition 3020 is implemented as adder 3120, and subtraction 3010 is implemented as subtracter 3110. Port p 2040 drives subtracter 3110. Port p' 2045 is driven by adder 3120. Port p 2040 and port p' 2045 are

partner ports. Partner ports are ports which represent the same signal, and thus frequently embody loop carry dependencies. Partner ports contain references to their partners. In the described embodiment, these references are implemented as pointers. Each port which has a partner contains a pointer to its partner port.

Figure 13 shows a constraint 1270 between adder node 2020, which is the producer for this LCD, and subtracter node 2010 which is the consumer of this LCD. The consumer and producer were identified in step 1110 of Figure 8. This constraint is created using the method of Figure 10. The starting templates are shown in Figure 13(a). First step 610 of Figure 10 adds placeholder node H 2060 to the template 1250 of subtracter node 2010. Because the initiation interval for the loop is 2, placeholder node H 2060 is constrained to be 2 cycles after subtracter node 2010 by template 1250. Next, step 620 creates constraint 1270, represented by an arrow, which constrains adder node 2020 to be at least one cycle before placeholder node H 2060. The modified templates and the new constraint are shown in Figure 13(b). The new constraint is then used to schedule the loop correctly using a method such as the one shown in Figure 9.

Figure 27 shows the add and subtract operations of Figure 12 scheduled into control steps by step 830 of Figure 6. For the sake of clarity, the other operations in the circuit are not shown. Two iterations of the loop are shown, to demonstrate how the schedule properly handles the loop carry dependency. Adder 3120 is scheduled so that its result is available before subtracter 3110 needs it in the next iteration of the loop.

Figure 14 shows the circuit created from the Verilog HDL source code of Figure 11 after scheduling. Block 3190 represents the representation of the FSM controller for this circuit stored in memory 104.

4.3 Using Placeholder-Nodes for I/O Dependencies

Loop pipelining must preserve the original order of all reads and writes to the same memory, signal, or port. The placeholder node method can be used to create constraints which ensure that I/O accesses in different iterations of the loop do not cross one another. Such constraints are created in step 1140 of Figure 8. The last I/O access to the same memory, signal, or port in a loop must occur simultaneously to or before the

first I/O access to that memory, signal or port in the next iteration of the loop. Specifically, reads of the same signal or port may occur simultaneously with reads in the next iteration of the loop, but not after. Writes to the same signal or port must occur before any read or write to the same signal or port in the next iteration of the loop. Reads and writes to the same memory must occur before any read or write to the same memory in the next iteration of the loop.

Thus, any last I/O access must occur within the initiation interval of the first I/O or memory access. To create this constraint, Event 1 of Figure 10 is set to be the first I/O access to a given memory, signal or port. Event 2 of Figure 10 is set to be the last I/O access to a given memory, signal or port. n is set to be the initiation interval of the loop, and c is set to be 0 or 1. Specifically, c is set to be 0 if Event 1 and Event 2 are signal or port reads. c is set to be 1 if Event 1 or Event 2 are signal or port writes, or memory reads or writes.

Figure 15 shows an example of Verilog source code for a loop 1530 with an I/O dependency between read 1510 and read 1520. Both read 1510 and read 1520 read the value of the same signal, x. Thus, read 1520 must be scheduled such that it occurs before read 1510 in the next iteration of the loop. In this example, a human designer has specified that this loop 1530 will be scheduled using an initiation interval of 1 and a latency of 3.

Figure 16 shows the GTech circuit 1500 which is created for loop 1530 of Figure 15. Circuit 1500 is output from step 810 of Figure 6. Read 1510 is implemented by read operation 3130. Read 1520 is implemented by read operation 3140. In this example, a human designer has specified that this loop will be pipelined with an initiation interval of 1 and a latency of 3.

Figure 17 shows a constraint between read node 1610, the first read of x in loop 1530, and read node 1620, the last read of x in loop 1530. Read node 1610 and read node 1620 were identified in step) 1130 of Figure 8. This constraint is created using the method of Figure 10. First step 610 adds placeholder node H 1760 to the template 1750 of read node 1610. Placeholder node H is constrained to be 1 cycle after read node

1610, because the initiation interval is 1, by template 1650. Next, step 620 creates constraint 1770, represented by an arrow, which constrains read node 1620 to be at least 0 cycles before, that is in the same cycle or after, placeholder node H 1760. Read node 1620 is constrained to be 0 cycles before placeholder node H 1760 because read node 1620 and read node 1610 are both signal reads, and as such are allowed to occur in the same control step. Constraint 1770 is then used to schedule the loop correctly using a method such as the one shown in Figure 9.

Figure 28 shows read operations on signal x of Figure 16 scheduled into control steps by step 830 of Figure 6. For the sake of clarity, the other operations in the circuit are not shown. Two iterations of the loop are shown, to demonstrate how the schedule properly handles the multiple signal reads. Read 3130 is scheduled so that it occurs simultaneously with read 3140 in the next iteration of the loop. Since simultaneous signal reads are allowed, this is a legal schedule.

Figure 18 shows the circuit created from the Verilog HDL source code of Figure 11 after scheduling.

5.0 Circuit Synthesis using Delayed Signal Assignment Information

Conventional design methodology uses a simulator to verify the correctness of a design both before and after it is synthesized. Conventional simulation systems, especially those systems performing behavioral synthesis, do not always yield identical cycle timing characteristics when HDL source code is simulated and when a synthesis output (a representation of a synthesized circuit) is simulated. It is advantageous for behavioral synthesis to be able to infer a circuit which will have the same cycle by cycle behavior during simulation as the simulation of the source HDL.

The source code of Figure 19(a) is written in the Verilog circuit specification language. The source code of Figure 19(b) is written in the VHDL circuit specification language. Both Verilog and VHDL are Hardware Description Languages (HDLs).

In Figure 19(a), the Verilog source code includes a signal assignment statement:

c<=#24x-p;

This statement includes a delay clause ("#24") indicating that a delay of twenty-

four time units, e.g., nanoseconds, should pass before the write operation is performed by the circuit that is to be generated. The delay clause is an example of delayed signal assignment information. Note that the inclusion of the delay clause in the HDL indicates a delay of the write operation only. The delay clause does not cause a delay in the performance of the subtraction operation. Similarly, in Figure 19(b), the VHDL source code includes a signal assignment statement:

c<=transportx-p after 24 ns;

This statement also contains a delay clause ("after 24 ns") indicating that a delay of twenty-four time units should occur in the generated circuit before the write operation is performed. This delay clause is a further example of delayed signal assignment information.

A circuit loop generated from the HDL source code of Figure 19(a) and Figure 19(b) will have an initiation interval of "2" because each source code example has two "wait" (or "posedge" or "negedge") statements within the loop. As discussed below, the delay clause in the source code causes the resulting loop to have a loop latency of "4". Figure 19(a) and Figure 19(b) are included for the purpose of example only. The present invention can use any appropriate type of source code (VHDL, Verilog, etc.) to represent a delay clause.

Figure 20 is a flowchart showing steps performed during translation step 810 of Figure 6 to generate a cdb. The exact placement of the steps of Figure 20 are not a part of the present invention and the steps also can be performed, for example, in the preprocessing step 820 of Figure 6. The input to Figure 20 is a representation of one of the source code examples of Figure 19(a) and Figure 19(b), such as a parse tree generated from the source code. The steps of Figure 20 are performed for each statement in the source code. The output of the translation step 810 and Figure 20 is a data flow graph (a "Gtech circuit") and a control flow graph (a "control data base" (cdb)). It will be understood by persons of ordinary skill in the art that the steps of Figure 20 and Figure 23 are performed by processor 109 of Figure 5, performing instructions stored in memory 104 of Figure 5.

In step 2002, the processor determines whether the current source code statement is a signal assignment statement (e.g., an assignment to a port using the "<=" operator) that includes a delay clause (e.g., "#24" in Verilog or "after 24 ns" in VHDL). If not, in step 2002, the processor performs standard processing for the node to build a node in the data flow graph. If the current source code statement includes a delay clause, then, in step 2004, the processor builds a write operation node in the data flow graph and annotates the node by adding an attribute indicating delayed signal assignment information to show that the write operation corresponding to the write operation node has a delay of, e.g., 24 nanoseconds (see node 2114 of Figure 21 and Figure 22).

Figure 21 shows an example of a data flow graph 2100 generated from one of the source code examples of Figure 19(a) and Figure 19(b) in accordance with the steps of Figure 20. A representation of data flow graph 2100 is stored in memory 104. Data flow graph 2100 includes as inputs a port x, a register p, and ports y and z. Each port has zero or more read operation nodes ("read op") 2102, 2104, 2106 associated therewith and each read operation node has an attribute indicating a port name (e.g., "port=`x`"). Respective ones of the inputs are input to a subtracter node 2110 and an adder node 2112. Subtracter node 2110 is connected to a write operation node 2114. Adder node 2112 is connected to a variable assignment node 2116. Output p' is input as p during successive iteration of the loop. Thus, the data flow graph of FIGURE 21 has seven nodes representing the data flow in the circuit to be synthesized.

In step 2008 of Figure 20, if there are more statements in the source code, control returns to step 2002. If all statements have been processed and a data flow graph (including signal delay attributes) has been generated for the source code, control passes to step 2012, where a control flow graph, such as that in Figure 22 is created.

Control graph 2200 of Figure 22 adds control information to nodes 2102, 2104, 2106, 2110, 2112, 2114, and 2116 indicating the order and conditions under which the data flow nodes are executed in the synthesized circuit. A representation of control graph 2200 is stored in memory 104 of Figure 5. The present invention preferably

operates in a "cycle fixed mode" in which each "wait" (or "posedge" or "negedge") statement in the source code indicates a new cycle in the synthesized circuit. Various processes for generating of control flow graphs are known to person of ordinary skill in the art and are described in High-Level Synthesis by Gajski et al.

In Figure 22, cnodes are used as "placeholder" nodes in the control graph to represent a collection of data flow nodes. Thus, cnode 2200 is associated with write operation node 2114 (including the signal delay attribute), read operation node 2102, and subtracter node 2110. The wait nodes in Figure 22 are used to represent the transitions between each cycle (or "cstep"). A wait node 2204 is used to mark the transition between the first cstep (cstep 0) and the second cstep (cstep 1). Wait node 2204 also has attributes indicating that it is based on a rising clock edge (due to the "posedge" statement in the source code) "Wait statements" (in VHDL source code) are treated similarly. Cnode 2206 (located in the second cstep) is associated with variable assignment node 2116, read operation node 2104, read operation node 2106, and adder node 2112. The control graph also includes a second wait node 2208 and a third cnode 2210.

As shown in Figure 7, the control flow graph is input to step 920, where a control data flow graph (CDFG) is created. The general procedure for creating a conventional CDFG is known to person of ordinary skill in the art and is described in High-Level Synthesis by Gajski et al. Figure 23 shows certain details of the process of creating a CDFG that relate to the delay clause of the present invention. An example CDFG is shown in Figure 24. The steps of Figure 23 are performed for each loop in the control flow graph. In step 2302, the processor sets a Wait.sub.-- count variable and a Max.sub.-- wait.sub.-- count variable in the memory 104 to an initial value of "0". In step 2304 the processor builds a "loop begin" node in the CDFG and assigns to it a cstep attribute value equal to "0".

Step 2306 is a first step in a loop performed by the processor for each cdb node. In step 2308, if the current cdb node is a cnode, control passes to step 2310, which is a first step in a loop performed for all data flow nodes associated with the current cdb

node. In step 2312, if a current data flow node is a write operation node having a delay clause (i.e., if the current data flow node represents a delayed signal assignment), control passes to step 2322.

In step 2322, a temp.sub.-- wait.sub.-- count variable is set to the current value of Wait.sub.-- count + a number of delay time units in the delayed signal assignment divided by the clock period (e.g., 0+24/6=4). A CDFG node is created and assigned to cstep temp.sub.-- wait.sub.-- count in step 2324. In step 2326, if temp.sub.-- wait.sub.-- count is greater than Max.sub.-- wait.sub.-- count, then in step 2328, Max.sub.-- wait.sub.-- count is set equal to temp.sub.-- wait.sub.-- count. Otherwise, control passes to step 2342. If, in step 2342, there are more data flow nodes associated with the current cdb node, then control passes to step 2310. Otherwise control passes to step 2336.

If, in step 2312, the current data flow nodes not a delayed signal assignment, the processor builds a standard CDFG node in step 2314 and assigns the created data flow node to cstep wait.sub.-- count in step 2316. If, in step 2318, wait.sub.-- count is greater than Max.sub.-- wait.sub.-- count, then Max.sub.-- wait.sub.-- count is assigned to wait.sub.-- count in step 2320. Control next passes to step 2342.

If, in step 2308, the current cdb node is not a cnode, then control passes to step 2330. If in step 2330 the current cdb node is a wait node, then wait.sub.-- count is incremented in step 2332 and control passes to step 2336. If, in step 2330, the current cdb node is not a wait node, then regular processing is performed to create a CDFG node in step 2334 and control passes to step 2336.

In step 2336, if there are more cdb nodes to process, then control passes to step 2306. Otherwise, a loop.sub.-- latency variable in memory 104 for the loop is assigned to Max.sub.-- wait.sub.-- count and an initiation interval variable for the loop is assigned to wait.sub.-- count in step 2338. In step 2340, the processor builds a "loop end" node in the CDFG and assigns it to cstep wait.sub.-- count.

The output of step 920 of Figure 7 is input to the scheduler, which uses the CDFG and the loop initiation interval and loop latency to schedule the nodes of the circuit being generated. In the described embodiment, all nodes except read/write

operation nodes can "float", i.e., can be moved between csteps by the scheduler to allow the scheduler to create an efficient circuit design. In the CDFG, these nodes are always assigned a cstep value equal to the initial cteps in which they appear in the HDL as a "suggestion" to the scheduler. It will be understood by persons of ordinary skill in the art that the CDFG of Figure 24 has been simplified for the sake of example and that the CDFG also includes, e.g., data flow arcs connecting the CDFG nodes that represent data flows in a similar manner to the data flows of Figure 21.

Figure 14 shows an example circuit synthesized from the CDFG of Figure 24. Figure 25 shows an example of placement of CDFG nodes in csteps without and with use of the delay clause. In the left column, which represents CDFG without the delay clause, CDFG nodes corresponding to write operation node 2114, read operation node 2109, and subtracter node 2110 are assigned to cstep 0. Similarly, CDFG nodes corresponding to adder node 2112, read operation node 2104, read operation node 2106, assignment node 2116 (and a CDFG loop.sub.-- end node) are assigned to a second cstep 1. Generation of this CDFG representation causes the synthesizer to generate a circuit that has different timing characteristics than the characteristics generated by the circuit synthesizer when the source code includes a delay clause. The right column of Figure 25 shows the assignment of CDFG nodes to cycles in accordance with the present invention. In this example, a write operation node corresponding to write operation node 2114 is moved into cstep 4 during the steps of FIGURE 23. This modification of the process to generate the CDFG (possible because of an addition of a signal delay attribute to the data graph 2100) allows the synthesis process to generate a circuit that has cycle level simulation behavior that is substantially identical to that of the cycle level simulation behavior of the source HDL.

Figure 26 shows an example of loop pipelining when the present invention is used. The figure shows an nth iteration of the loop and an n+1st iteration of the loop over time. As can be seen in the figure, the initial interval of successive iterations of the loop is equal to a number of wait statements (or "posedge" or "negedge" statements). The loop latency, is equal to the longest cycle delay from the beginning of the loop to a

latest operation. The throughput of the pipelined loop is not decreased by use of delayed signal assignments. In general, the scheduler will schedule a circuit having the CDFG of Figure 24 as a pipelined circuit because the loop latency is longer than the initiation interval.

In summary, use of delayed signal assignments allows behavioral synthesis to infer circuits with pipelined loops which have cycle level simulation behavior which matches that of the source HDL. Pipelined loops may include loop carry dependencies and/or I/O and/or memory accesses which must be scheduled correctly. The use of a placeholder node within a template is an efficient representation of such scheduling constraints.

WHAT IS CLAIMED IS:

1. A method performed by a data processing system having a memory, comprising the steps of:

parsing a text description of a circuit, said text description stored in the memory, said text description including a loop with a delayed signal assignment having a delay value;

translating said text description into a digital circuit representation in said memory, said digital circuit representation including a pipeline; and

setting a latency of said pipeline equal to said delay value.

2. The method of claim 1, wherein said loop further includes N wait statements, where N is greater than zero, said method further comprising the step of setting an initiation interval of said pipeline equal to N.

3. The method of claim 1, wherein said text description is written in Verilog and said delayed signal assignment uses a Verilog "#" operator.

4. The method of claim 3, wherein said wait statements use Verilog "@posedge" statements.

5. The method of claim 3, wherein said wait statements use Verilog "@negedge" statements.

6. The method of claim 1, wherein said text description is written in VHDL, said delayed signal assignment uses a VHDL "after" clause, and said wait statements use VHDL "wait" statements.

7. A method, performed by a data processing system having a memory of building a

digital circuit representation including a pipeline in the memory from a textual description of a loop, comprising the steps of:

identifying a loop carry dependency in said loop;

identifying a producer operation of said loop carry dependency;

identifying a consumer operation of said loop carry dependency;

determining a number, n, of cycles within which said producer operation must be scheduled after said consumer operation;

instantiating a placeholder node in said memory;

node-locking said placeholder node so that it must be scheduled n cycles after said consumer operation; and

constraining said producer operation to be scheduled before said placeholder node.

8. The method of claim 7, wherein the step of node-locking said placeholder node further comprises the step of creating a template structure in said memory which includes said placeholder node and said consumer operation.

9. The method of claim 8,

wherein said producer operation is included in a second template structure in said memory, and

wherein the step of constraining said producer operation further comprises the step of constraining said second template structure to be scheduled before said template structure.

10. The method of claim 7, wherein n is equal to an initiation interval of said pipeline multiplied by a number of iterations of said loop which execute before data produced by said producer is consumed by said consumer.

11. A method, performed by a data processing system having a memory, of building

a digital circuit representation in said memory, said digital circuit representation including a pipeline derived from a textual description of a loop, said method comprising the steps of:

identifying an access dependency of said loop;

identifying a first access operation of said access dependency;

identifying a second access operation of said access dependency;

determining a number, n, of cycles within which said second access operation must be scheduled after said first access operation;

instantiating a placeholder node in said memory;

node-locking said placeholder node so that it must be scheduled n cycles after said first access operation; and

constraining a scheduling order of said second access operation and said placeholder node.

12. The method of claim 11,

wherein said first access operation is chosen from the group of access operations including a memory read, a memory write, a signal write and a port write,

said second access operation is chosen from the group of access operations including a memory read, a memory write, a signal read, a signal write, a port read and a port write, and

the step of constraining said scheduling order of said second access operation and said placeholder node further includes the step of forcing said second access operation to be scheduled before said placeholder node.

13. The method of claim 11,

wherein said first access operation is chosen from the group of access operations including a memory read, a memory write, a signal read, a signal write, a port read and a port write,

said second access operation is chosen from the group of access operations

including a memory read, a memory write, a signal write and a port write, and

the step of constraining said scheduling order of said second access operation and said placeholder node further includes the step of forcing said second access operation to be scheduled before said placeholder node.

14. The method of claim 11,

wherein said first access operation is chosen from the group of access operations including a signal read and a port read,

said second access operation is chosen from the group of access operations including a signal read and a port read, and

the step of constraining said scheduling order of said second access operation and said placeholder node further includes the step of forcing said second access operation to be scheduled simultaneous with, or before said placeholder node.

15. The method of claim 11, wherein the step of constraining said scheduling order of said second access operation and said placeholder node further includes the step of forcing said second access operation to be scheduled before said placeholder node.

16. The method of claim 11, wherein the step of node-locking said placeholder node further includes the step of creating a template which includes said placeholder node and said first access operation.

17. The method of claim 11, wherein n is equal to an initiation interval of said pipeline multiplied by a number of iterations of said loop which execute between said first access operation and said second access operation.

18. A system for building, in a memory, a digital circuit representation which implements the behavior of a text description in said memory, said system having a processor coupled to a memory unit wherein said processor is programmed to perform

logic processing, said system comprising:

parsing logic for parsing said text description into a parsed text description, said text description including a loop with a delayed signal assignment having a delay value;

translating logic for translating said parsed text description into said digital circuit representation, said digital circuit including a pipeline; and

latency setting logic for setting a latency value of said pipeline to be said delay value of said delayed signal assignment.

19. A system as described in claim 18, wherein said pipeline implements said loop.

20. A system as described in claim 19, wherein said loop further includes a number, n, of wait statements, said system further comprising initiation interval setting logic for setting an initiation interval of said pipeline to be equal to n.

21. A computer program product comprising:

a computer usable medium having computer readable code embodied therein for building a digital circuit representation from a text description of a digital circuit, the computer program product comprising:

computer readable program code devices configured to cause a computer to effect parsing said text description, said text description including a loop with a delayed signal assignment having a delay value;

computer readable program code devices configured to cause a computer to effect translating said text description into said digital circuit representation including a pipeline; and

computer readable program code devices configured to cause a computer to effect setting a latency of said pipeline equal to said delay value.

22. The computer program product of claim 21 wherein said loop further includes N wait statements, where N is greater than zero, said computer program product further

comprising computer readable program code devices configured to cause a computer to effect setting an initiation interval of said pipeline equal to N.

23. A method performed by a data processing system having a memory, comprising the steps of:

parsing a text description of a circuit, said text description stored in the memory, said text description including a loop with N wait statements, where N is greater than zero;

translating said text description into a digital circuit representation in said memory, said digital circuit representation including a pipeline; and

setting an initiation interval of said pipeline equal to N.

24. The method of claim 23, wherein the wait statements are VHDL wait statements.

25. The method of claim 23, wherein the wait statements are Verilog HDL @posedge statements.

26. The method of claim 23, wherein the wait statements are Verilog HDL @negedge statements.

27. A system for building, in a memory, a digital circuit representation which implements the behavior of a text description in said memory, said system having a processor coupled to a memory unit wherein said processor is programmed to perform logic processing, said system comprising:

parsing logic for parsing said text description into a parsed text description, said text description including a loop with N wait statements, where N is greater than zero;

translating logic for translating said parsed text description into said digital circuit representation, said digital circuit including a pipeline; and

initiation interval setting logic for setting an initiation interval of said pipeline equal to N.

28. The system of claim 27, wherein the wait statements are VHDL wait statements.

29. The system of claim 27, wherein the wait statements are Verilog HDL @posedge statements.

30. The system of claim 27, wherein the wait statements are Verilog HDL @negedge statements.

31. A computer program product comprising a computer usable medium having computer readable code embodied therein for building a digital circuit representation from a text description of a digital circuit, the computer program product comprising:

computer readable program code devices configured to cause a computer to effect parsing said text description, said text description including a loop with N wait statements, where N is greater than zero;

computer readable program code devices configured to cause a computer to effect translating said text description into said digital circuit representation including a pipeline; and

computer readable program code devices configured to cause a computer to effect setting an initiation interval of said pipeline equal to N.

32. The method of claim 31, wherein the wait statements are VHDL wait statements.

33. The method of claim 31, wherein the wait statements are Verilog HDL

@posedge statements.

34. The method of claim 31, wherein the wait statements are Verilog HDL @negedge statements.

Abstract

METHODS FOR AUTOMATICALLY PIPELINING LOOPS

A method and an apparatus for creating a representation of a circuit with a pipelined loop from an HDL source code description. It infers a circuit including a pipelined loop which has cycle level simulation behavior matching that of the source HDL. Loop carry dependencies and memory and signal I/O accesses within the loop are scheduled correctly.

THIS PAGE BLANK (USPTO)

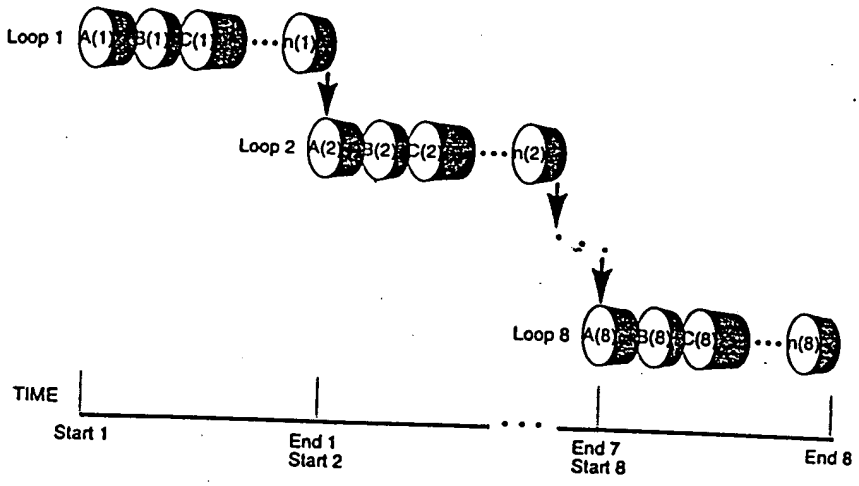

Loop 1
A(1)
B(1)
C(1)
n(1)
Loop 2
A(2)
B(2)
C(2)
n(2)
Loop 8
A(8)
B(8)
C(8)
n(8)
TIME
Start 1
End 1
Start 2
End 7
Start 8
End 8

Figure 1

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 1 of 26

THIS PAGE BLANK (USPTO)

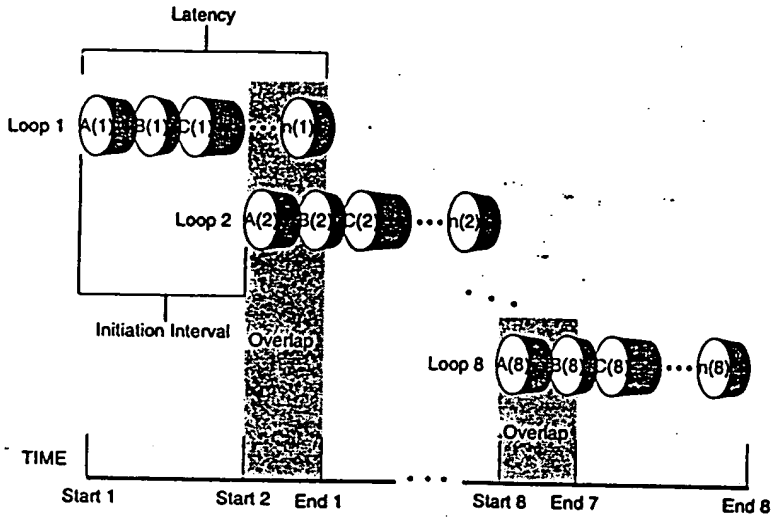

Latency
Loop 1
A(1)
B(1)
C(1)
n(1)
Loop 2
A(2)
B(2)
C(2)
n(2)
Initiation Interval
Overlap
Loop 8
A(8)
B(8)
C(8)
n(8)
Overlap
TIME
Start 1
Start 2
End 1
Start 8
End 7
End 8

Figure 2

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 2 of 26

THIS PAGE BLANK (USPTO)

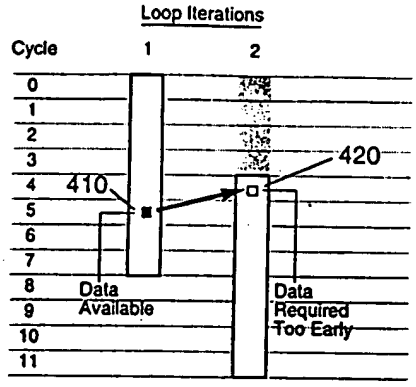


Figure 3 (a)

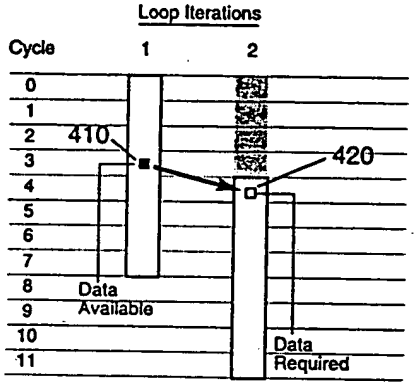


Figure 3 (b)

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USPTO)

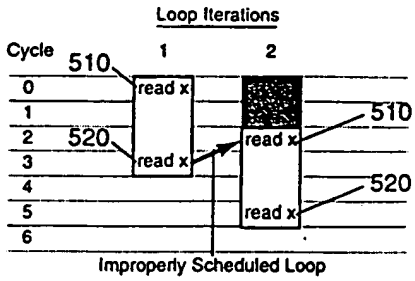


Figure 4 (a)

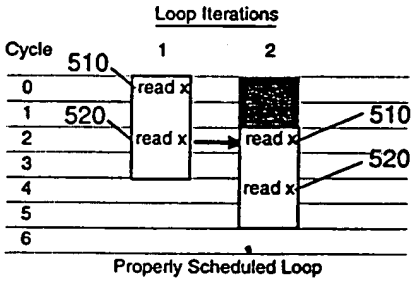


Figure 4 (b)

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USPTO)

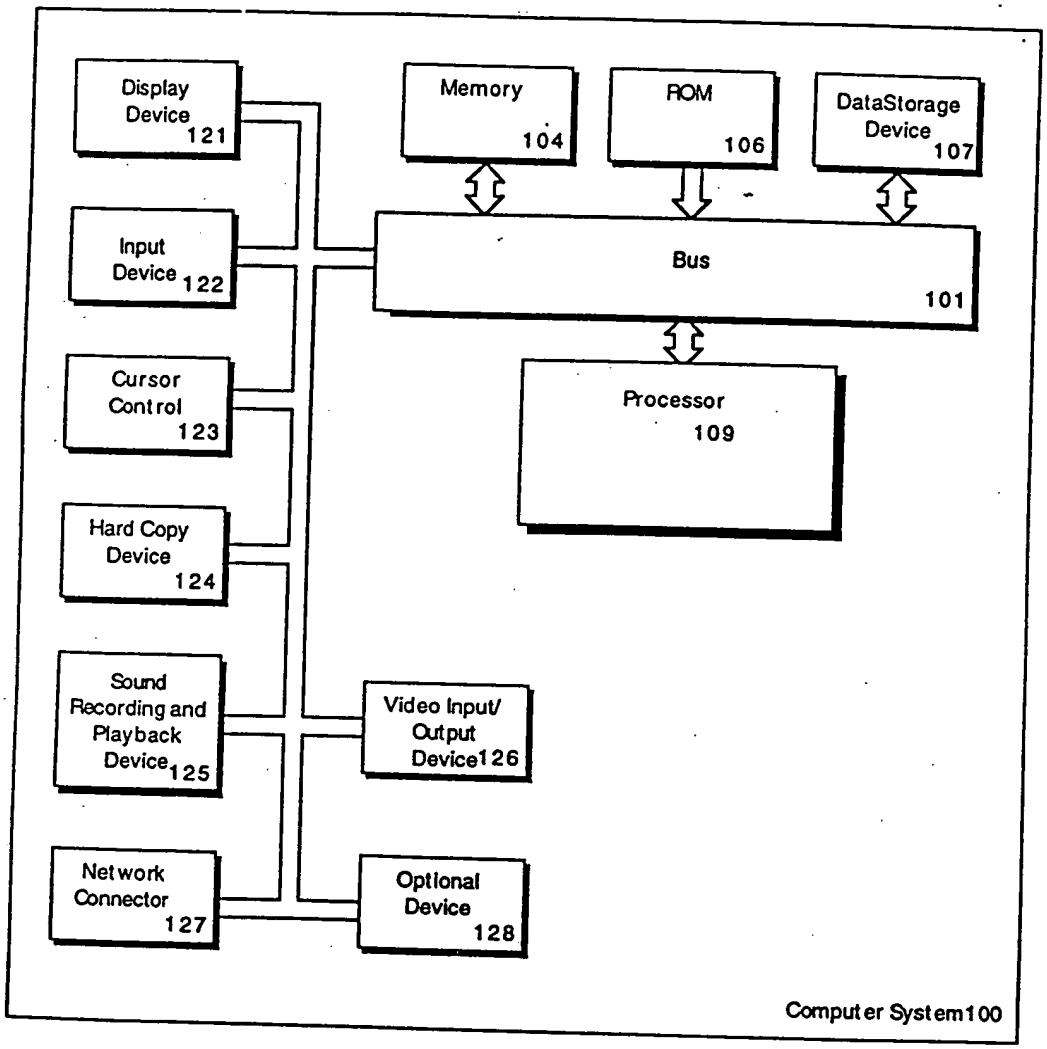

Display Device 121
Memory 104
ROM 106
DataStorage Device 107
Input Device 122
Bus 101
Cursor Control 123
Processor 109
Hard Copy Device 124
Sound Recording and Playback Device 125
Video Input/ Output Device126
Network Connector 127
Optional Device 128
Computer System100

Figure 5

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 5 of 26

THIS PAGE BLANK (USPTO)

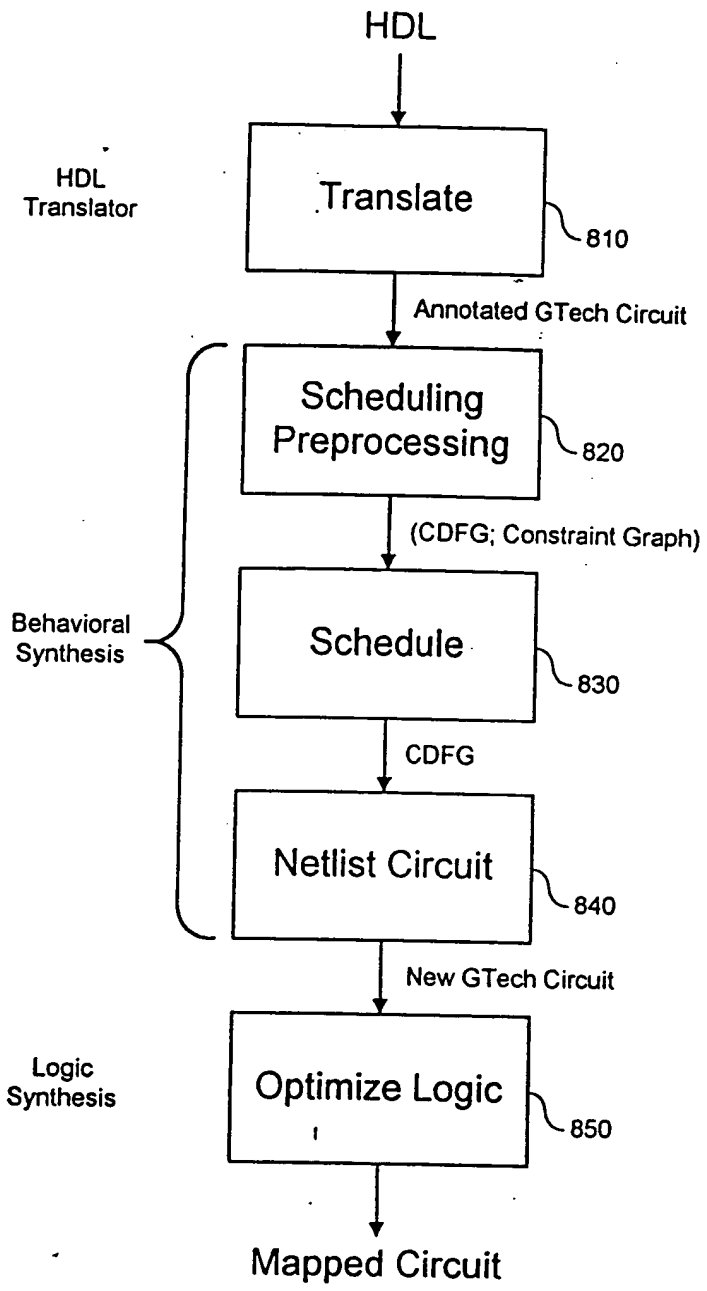


Figure 6 Synthesis with Scheduling

Atty. Docket No. 4000/.
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 6 of 26

THIS PAGE BLANK (USPTO)

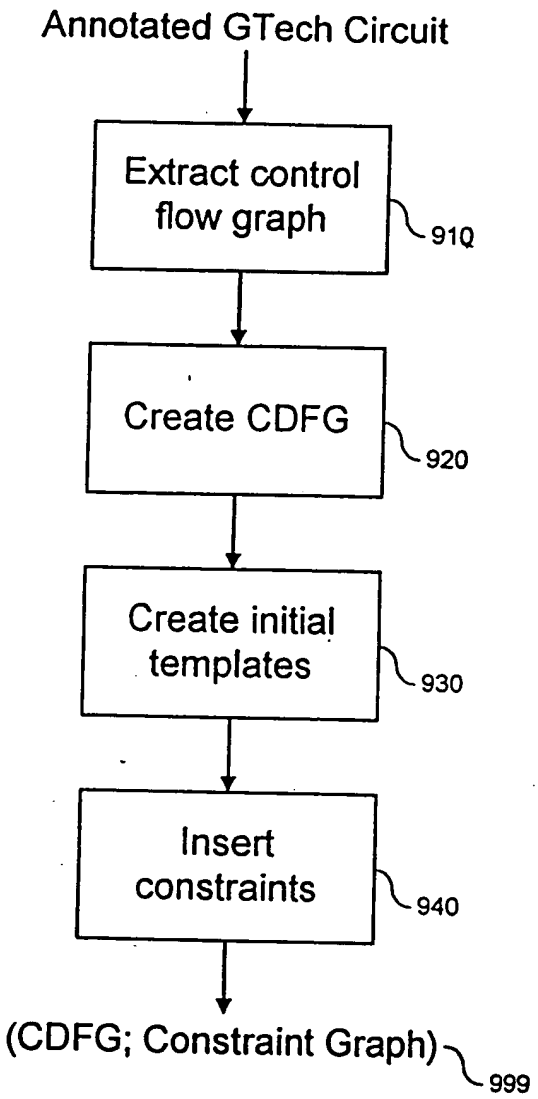


Figure 7 Scheduling Preprocessing

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 7 of 26

THIS PAGE BLANK (USPTO)

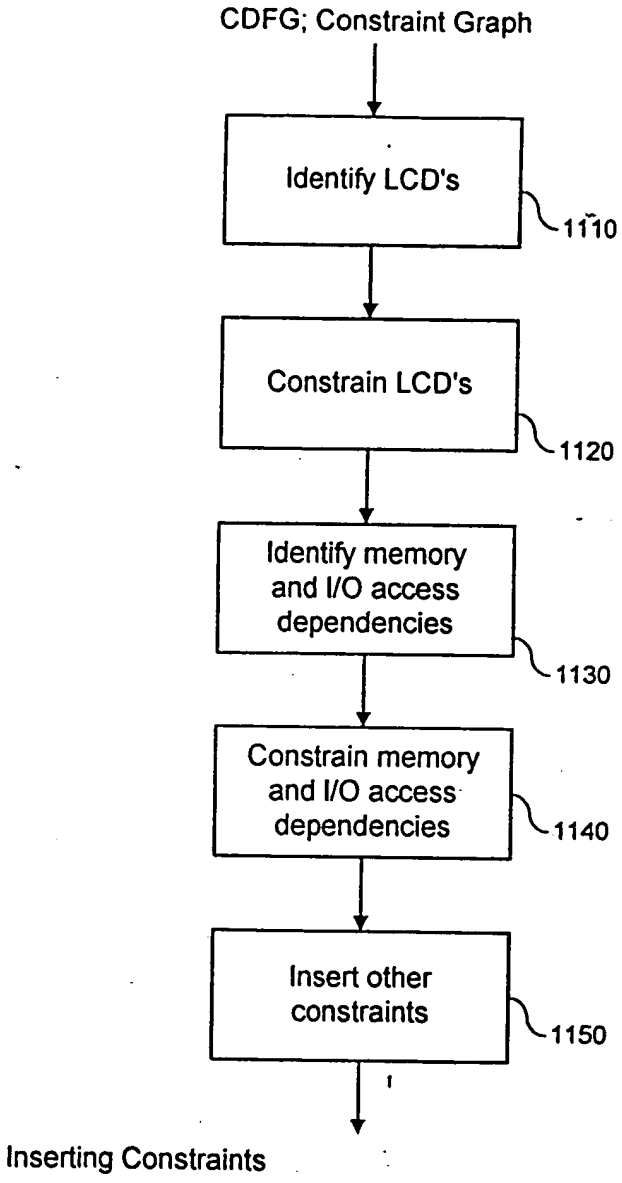

CDFG; Constraint Graph
Identify LCD's
1110
Constrain LCD's
1120
Identify memory and I/O access dependencies
1130
Constrain memory and I/O access dependencies
1140
Insert other constraints
1150
Inserting Constraints

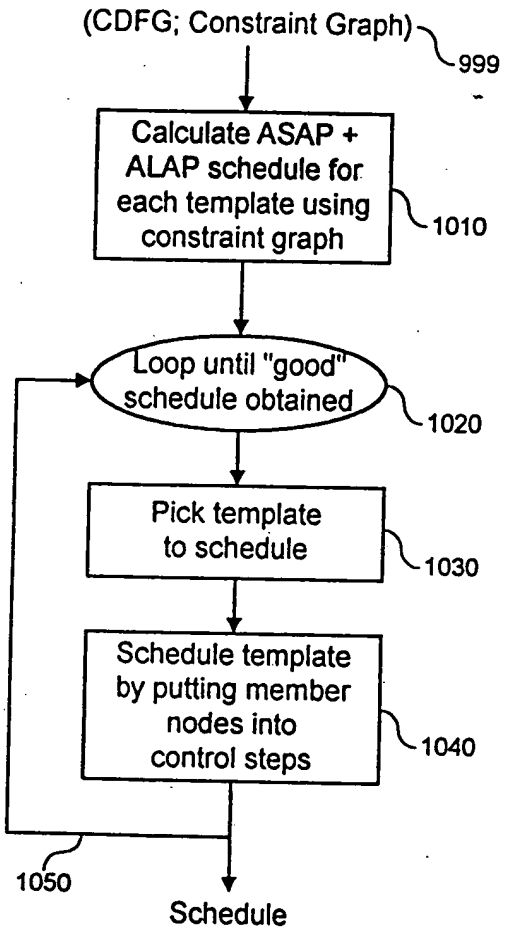
Figure 8

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 8 of 26

THIS PAGE BLANK (USPTO)



Scheduling Using Templates

Figure 9

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 9 of 26

THIS PAGE BLANK (USPTO)

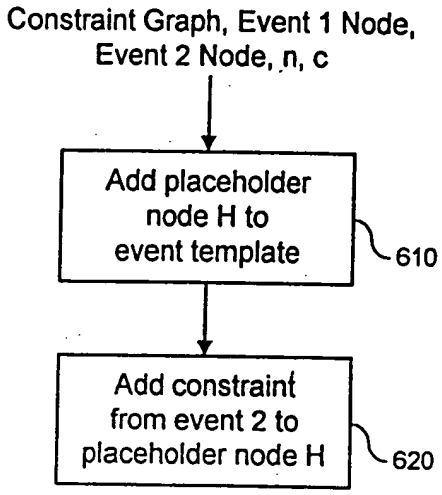

Constraint Graph, Event 1 Node, Event 2 Node, n, c
Add placeholder node H to event template
610
Add constraint from event 2 to placeholder node H
620

Figure 10

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 10 of 26

THIS PAGE BLANK (USPTO)

```
module loopex8 ( c, x, y, z, clock);
input [1:0] x, y, z;
   input clock ;
   output [2:0] c;
   reg [2:0] c;
   reg [2:0] p;

   always begin                                    3030

        forever begin : theloop                              3010

                c <= x - p ;

                @(posedge clock) ;                          3020

                p = y + z ;

                @(posedge clock) ;

        end

   end

endmodule
```

Figure 11

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 11 of 26

THIS PAGE BLANK (USPTO)

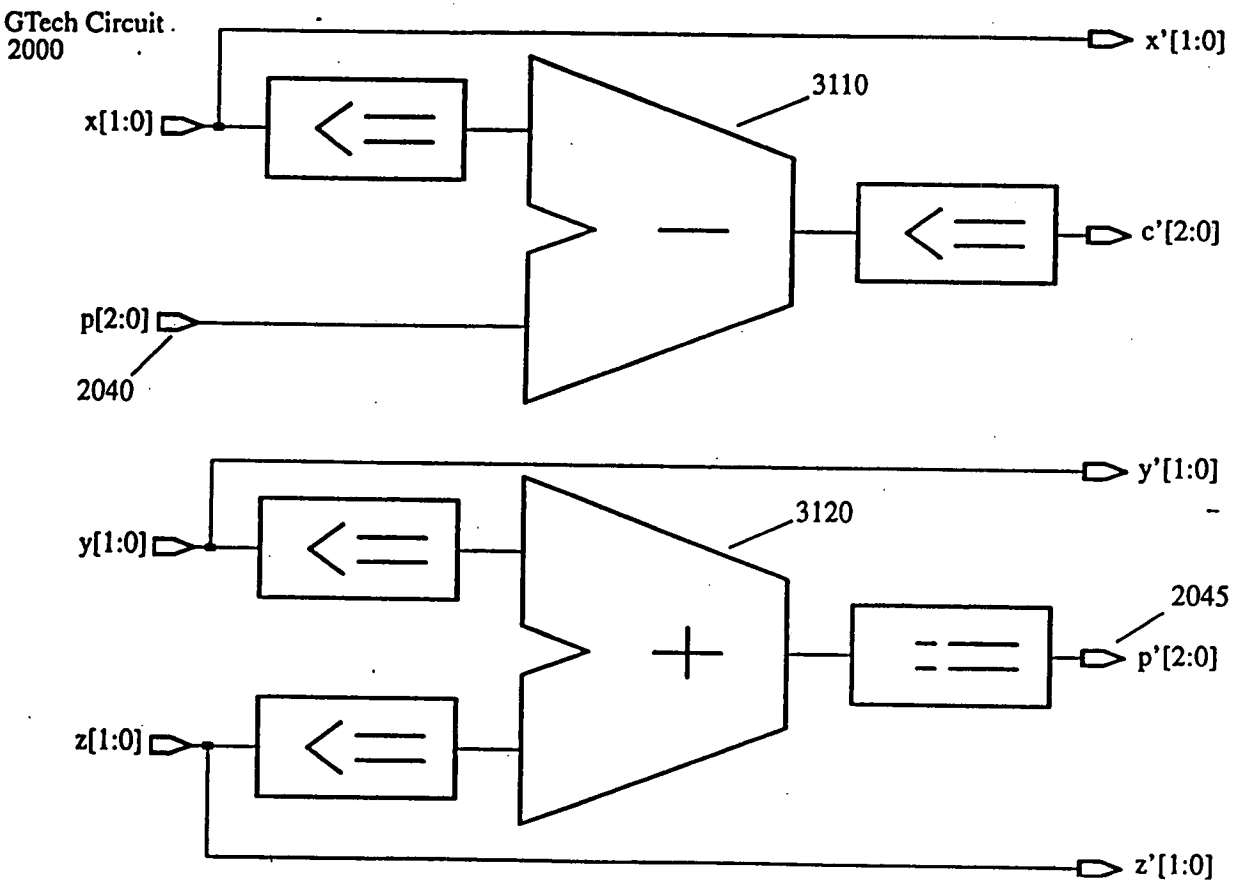
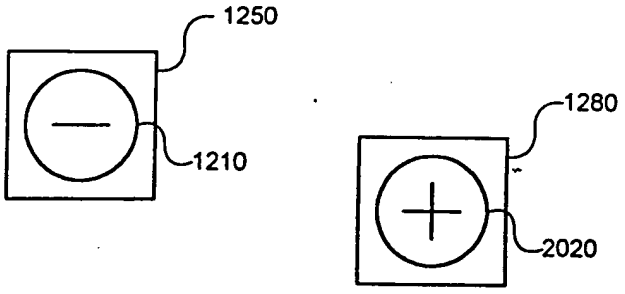

GTech Circuit
2000
x'[1:0]
x[1:0]
3110
c'[2:0]
p[2:0]
2040
y'[1:0]
y[1:0]
3120
2045
p'[2:0]
z[1:0]
z'[1:0]

Figure 12

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 12 of 26

THIS PAGE BLANK (USPTO)



n = 2

Figure 13a

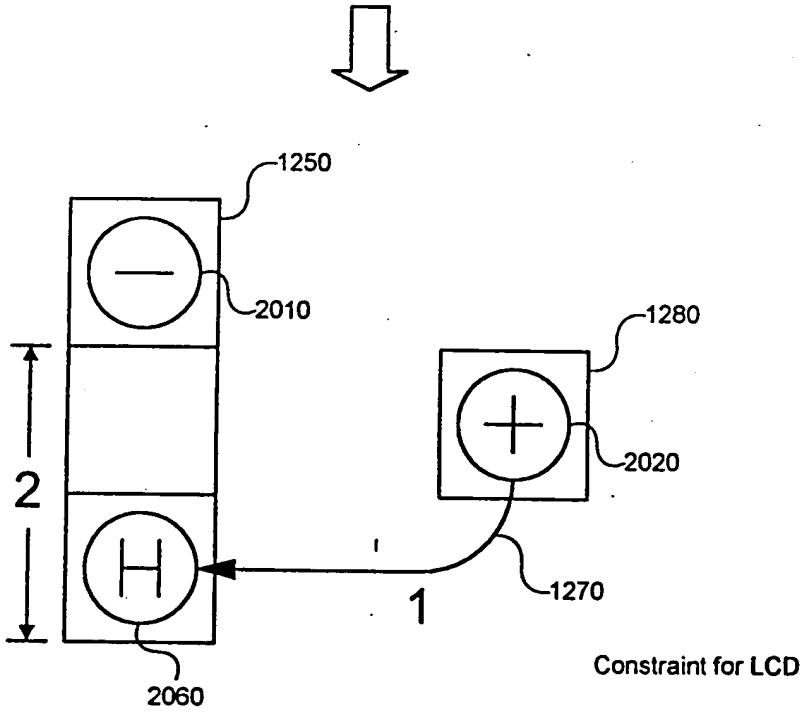


Figure 13b

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 13 of 26

THIS PAGE BLANK (USPTO)

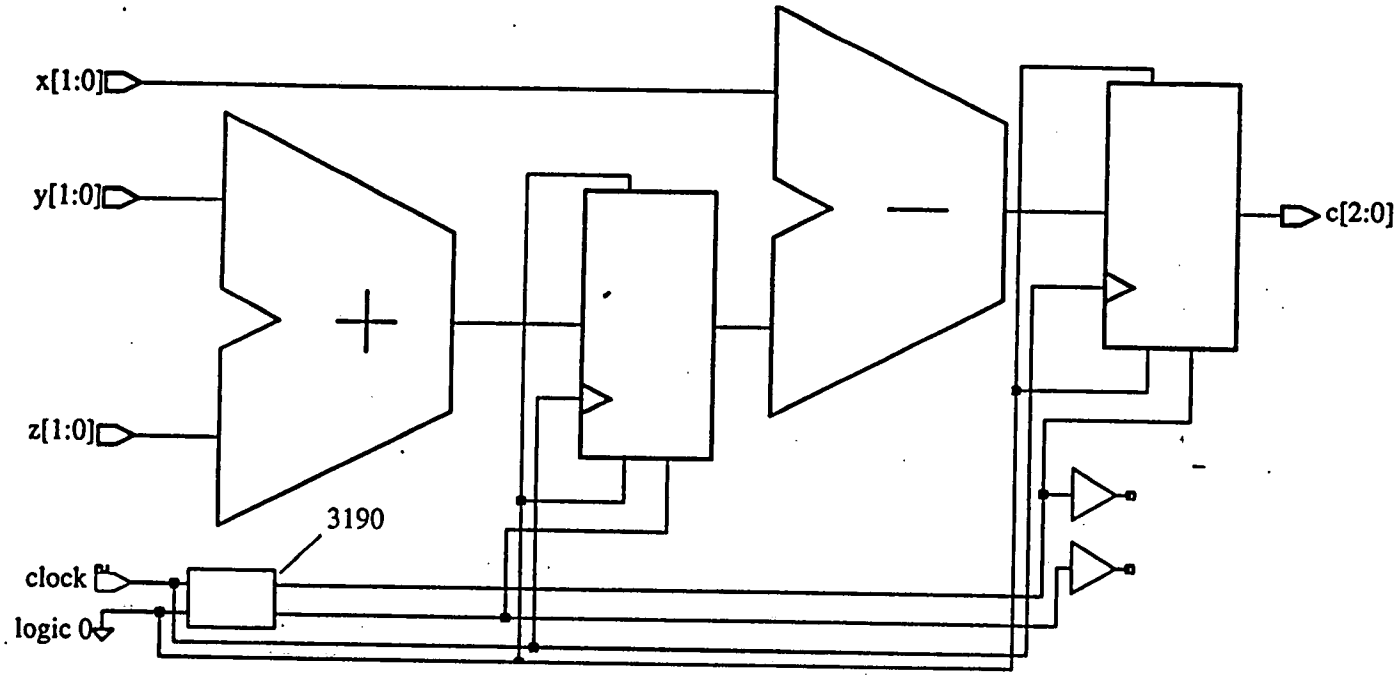


Figure 14

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USPTO)

```
module write4 ( w, x, clock);

  input [15:0] x ;
  input clock ;
  output [31:0] w;
  reg [32:0] w;
  reg [15:0] x1 ;
  reg [15:0] x2 ;

  always begin                          1530

      forever begin : writeloop         1530

            x1 <= x ;

            @(posedge clock) ;          1530

            x2 <= x ;

            w <= x1 * x2 ;

      end

  end

endmodule
```

Figure 15

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 15 of 26

THIS PAGE BLANK (USPTO)

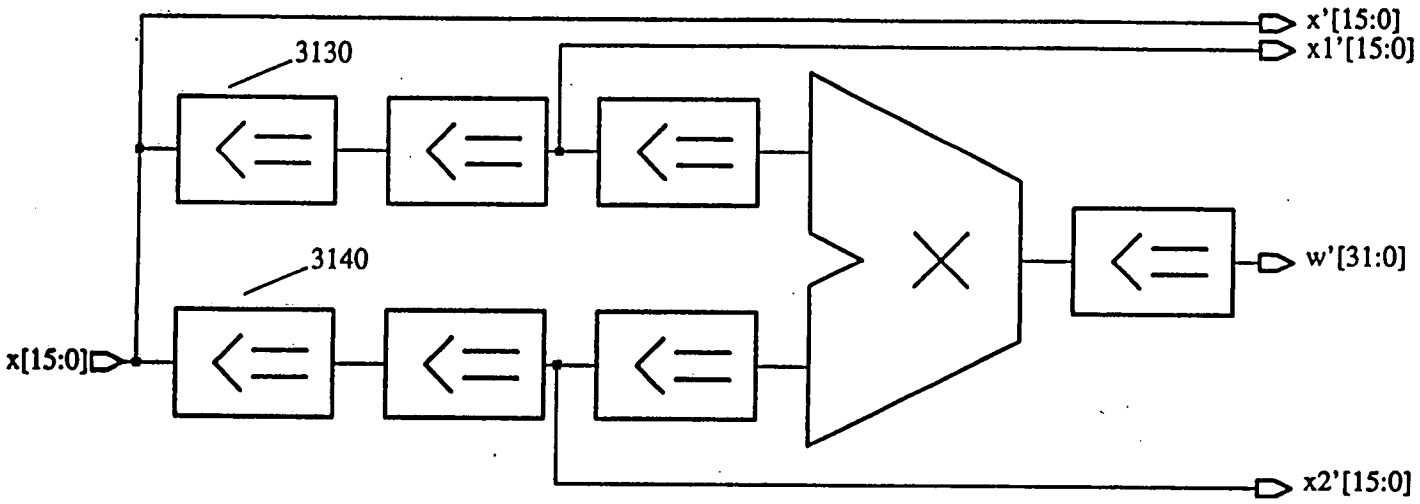


Figure 16

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 16 of 26

THIS PAGE BLANK (USPTO)

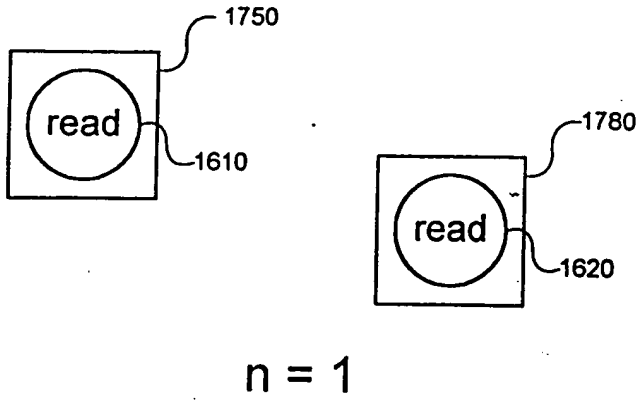


Figure 17a

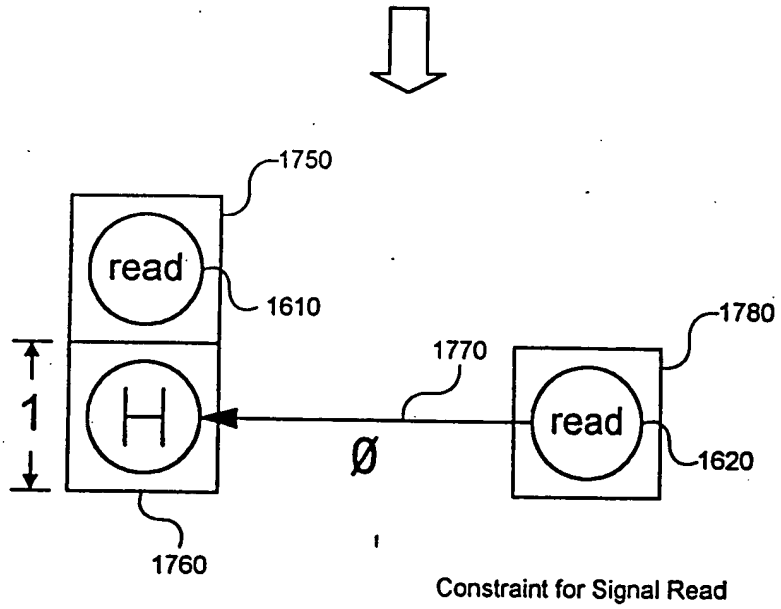


Figure 17b

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 17 of 26

THIS PAGE BLANK (USPTO)

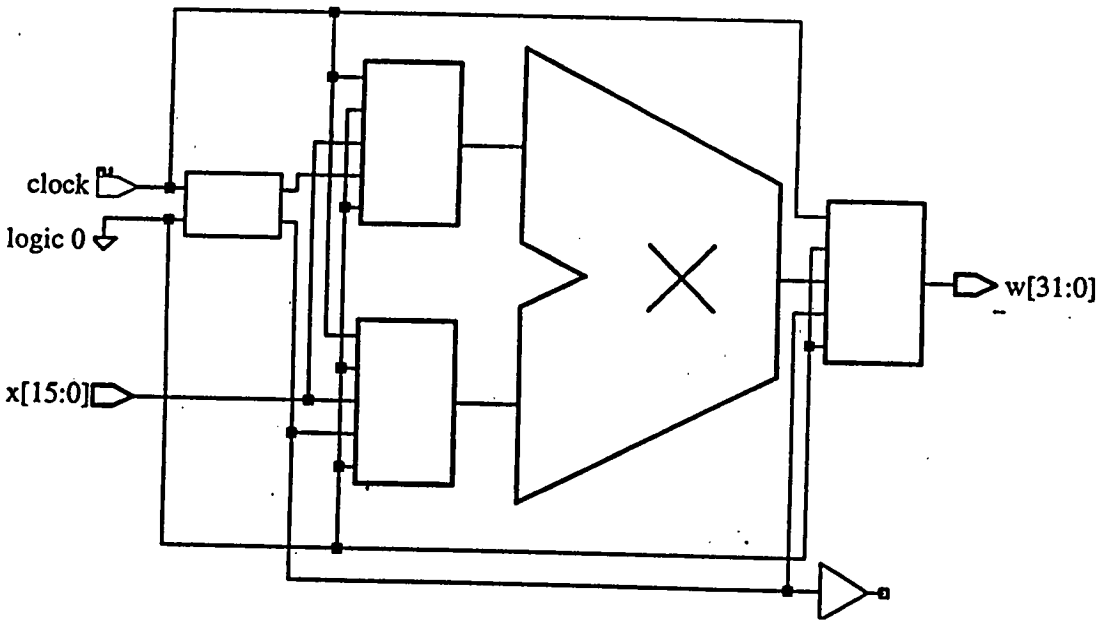


Figure 18

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 18 of 26

THIS PAGE BLANK (USPTO)

```
module after1 ( c, x, y, z, clock);

  input [1:0] x, y, z;
  input clock ;
  output [2:0] c;
  reg [2:0] c;
  reg [2:0] p;

  always begin

        @(posedge clock) ;

        forever begin
              c <= #24 x - p ;

              @(posedge clock) ;

              p = y + z ;

              @(posedge clock) ;
        end
  end
endmodule
```

Figure 19 (a)

```
entity after1 is
  port(
        c : out integer range 0 to 7;
        x, y, z : in integer range 0 to 3;
        clock : in bit
  );
end after1;

architecture behavioral of after1 is begin
  process
        variable p : integer range 0 to 7;
  begin
        wait until clock'event and clock = '1';

        loop

          c <= transport x - p after 24 ns;

          wait until clock'event and clock = '1';

          p := y + z;

          wait until clock'event and clock = '1';

        end loop;

  end process;
end behavioral;
```

Figure 19 (b)

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 19 of 26

THIS PAGE BLANK (USPTO)

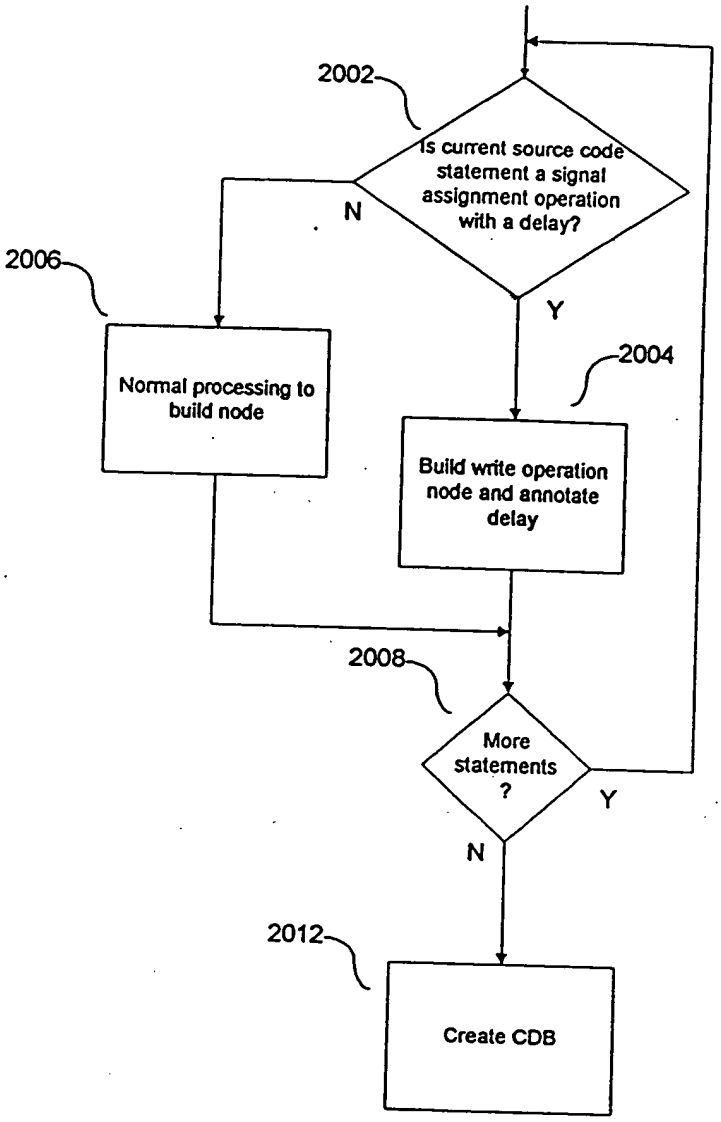

2002
Is current source code statement a signal assignment operation with a delay?
N
Y
2006
Normal processing to build node
2004
Build write operation node and annotate delay
2008
More statements ?
Y
N
2012
Create CDB

Fig. 20

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 20 of 26

THIS PAGE BLANK (USPTO)

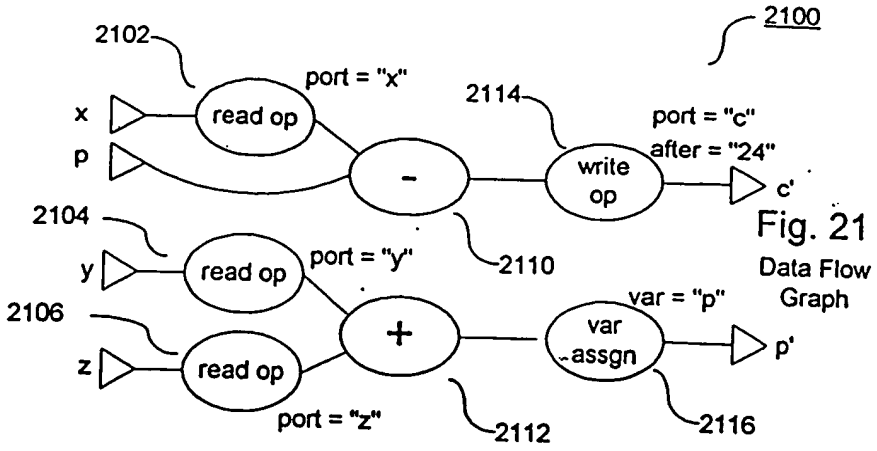


Fig. 21
Data Flow Graph

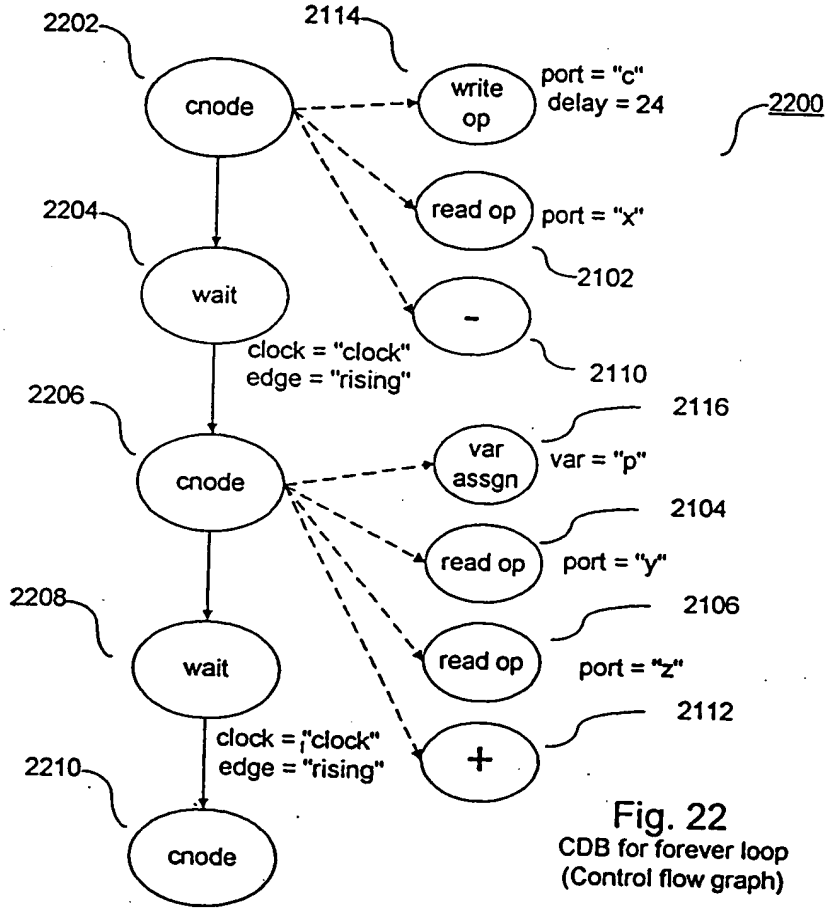


Fig. 22
CDB for forever loop
(Control flow graph)

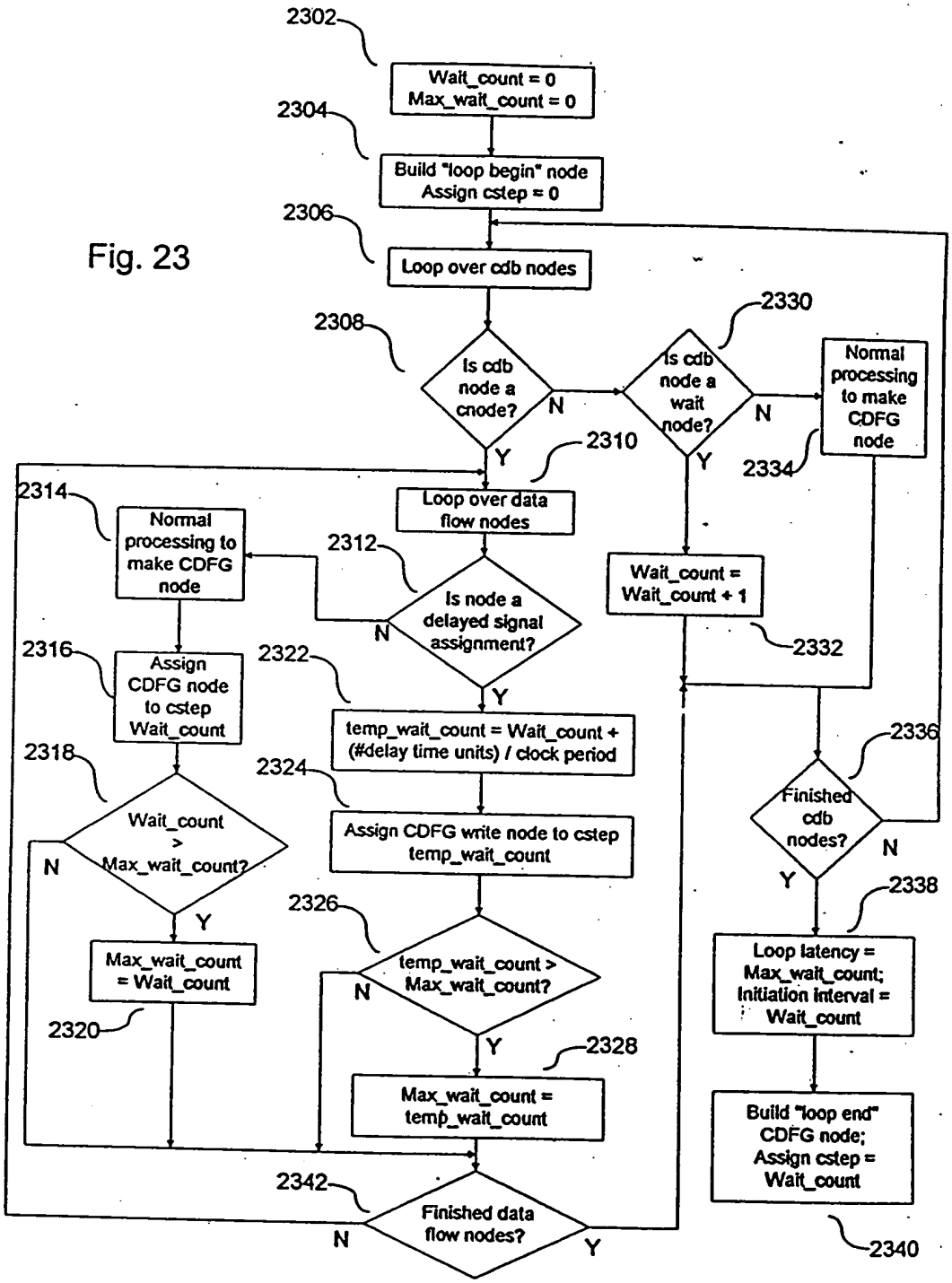
Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 21 of 26

THIS PAGE BLANK (USPTO)

Fig. 23


2302
Wait_count = 0
Max_wait_count = 0
2304
Build "loop begin" node
Assign cstep = 0
2306
Loop over cdb nodes
2308
Is cdb node a cnode?
N
Y
2330
Is cdb node a wait node?
N
Y
Normal processing to make CDFG node
2334
2310
Loop over data flow nodes
2314
Normal processing to make CDFG node
2312
Is node a delayed signal assignment?
N
Y
Wait_count = Wait_count + 1
2332
2316
Assign CDFG node to cstep Wait_count
2322
temp_wait_count = Wait_count + (#delay time units) / clock period
2318
Wait_count > Max_wait_count?
N
Y
2324
Assign CDFG write node to cstep temp_wait_count
Max_wait_count = Wait_count
2320
2326
temp_wait_count > Max_wait_count?
N
Y
2328
Max_wait_count = temp_wait_count
2336
Finished cdb nodes?
N
Y
2338
Loop latency = Max_wait_count; Initiation interval = Wait_count
Build "loop end" CDFG node; Assign cstep = Wait_count
2340
2342
Finished data flow nodes?
N
Y

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USPTO)

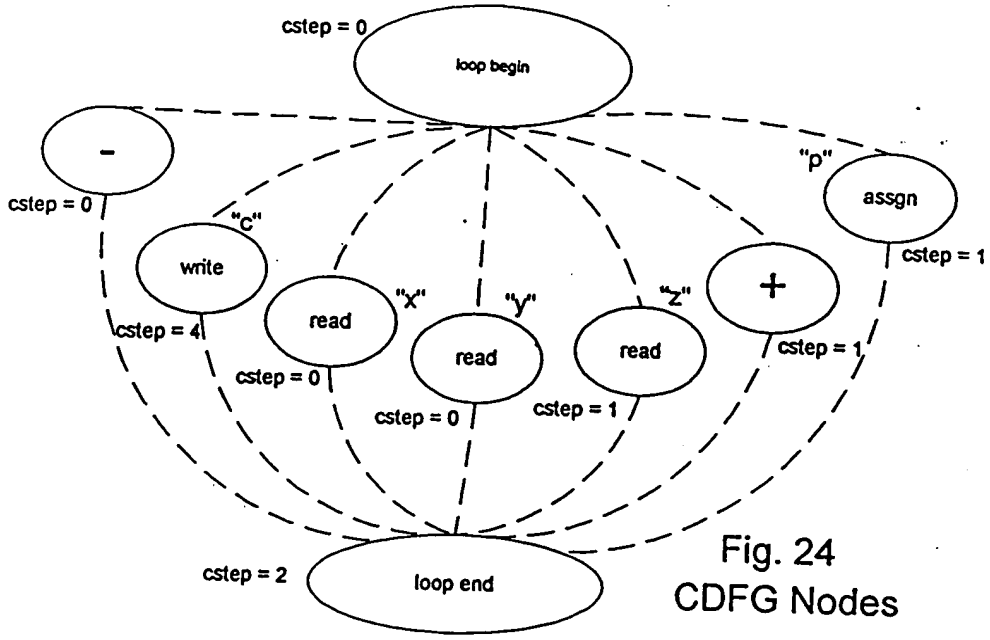


Fig. 24
CDFG Nodes

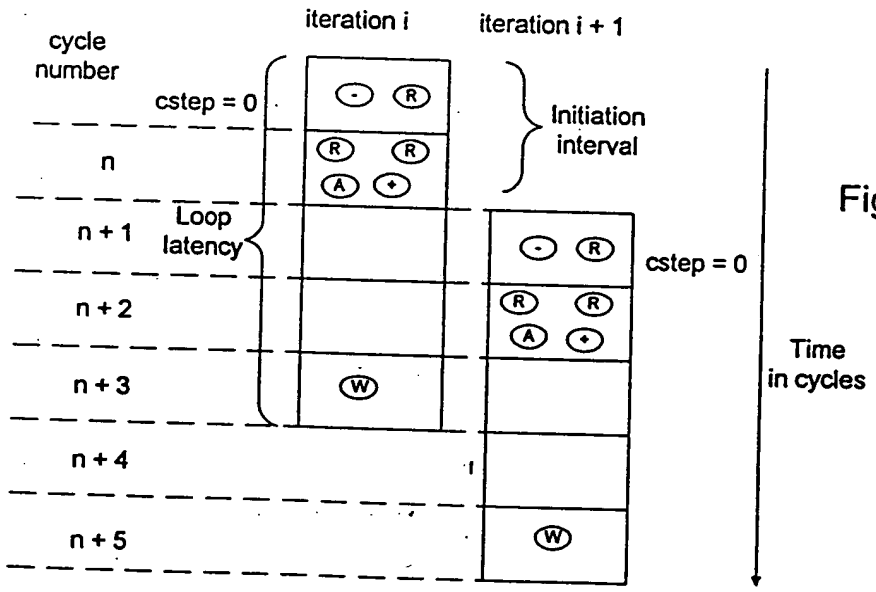


Fig. 26

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USPTO)

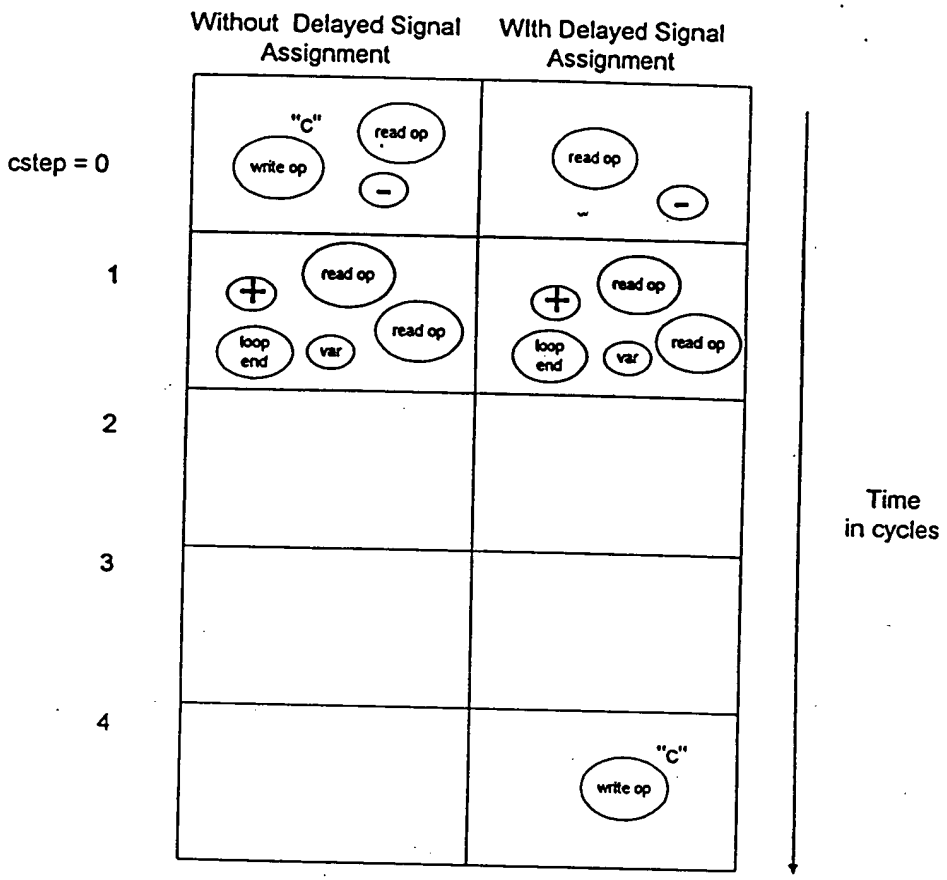

Without Delayed Signal Assignment
With Delayed Signal Assignment
cstep = 0
1
2
3
4
"c"
write op
read op
read op
read op
loop end
var
read op
read op
loop end
var
read op
"c"
write op
Time in cycles

Fig. 25

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USPTO)

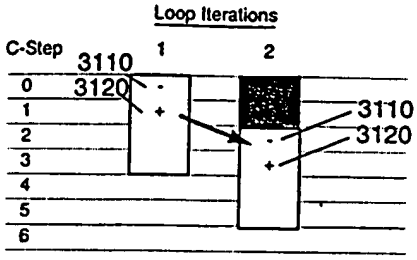


Figure 27

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 25 of 26

THIS PAGE BLANK (USPTO)

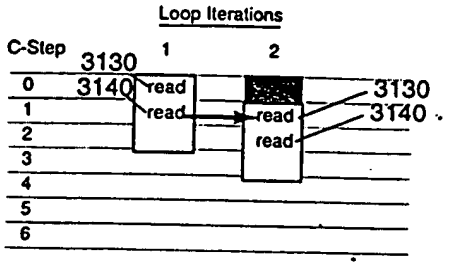

Loop Iterations
C-Step
1
2
3130
3140
read
read
read
read
3130
3140
0
1
2
3
4
5
6

Figure 28

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 26 of 26

Appendix A

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX A
Sheet 1 of 7

Scheduling using Behavioral Templates

Tai Ly, David Knapp, Ron Miller, Don MacMillen
Synopsys Inc.
700B E. Middlefield Road
Mountain View, CA USA 94043

Abstract: This paper presents the idea of "behavioral templates" in scheduling. A behavioral template locks several operations into a relative schedule with respect to one another. This simple construct proves powerful in addressing: (1) timing constraints, (2) sequential operation modeling, (3) pre-chaining of certain operations, and (4) hierarchical scheduling. We present design examples from industry to demonstrate the importance of these issues in scheduling.

1.0 Introduction

The task of scheduling [4] is to sequence nodes in a control and data flow graph (CDFG) by assigning each node to a control step (*cstep*). We present the idea of behavioral templates, and describe how we use behavioral templates to address several issues that arise when applying scheduling to commercial designs. For the purpose of this paper, we assume timing constrained scheduling [5].

A behavioral template specifies a relative scheduling among its member CDFG nodes. It is a template in the sense that its member nodes can be treated as a single scheduling unit by assigning the starting cstep for the template. It is behavioral in the sense that it specifies a scheduling pattern as opposed to, for example, a structural pattern [8]. We extend scheduling algorithms to handle behavioral templates by recasting the task of scheduling as that of assigning templates to csteps.

Although a simple idea, behavioral templates provide a powerful way to address four issues in scheduling:

1. **Timing constraints.** We use behavioral templates to impose fixed and maximum timing constraints. This is more efficient than using precedence edges alone because an entire sequence of nodes is considered at once when scheduling one template.
2. **Multi-cycle operations.** To enable scheduling of complex multi-cycle operations, we use multiple CDFG nodes locked in a behavioral template to model the cycle-by-cycle I/O and resource requirements of such operations.
3. **Logic and bit-manipulation operations.** We use behavioral templates to force certain chaining of logic and bit-manipulation operations to save register costs. This reduces the scheduling design space, and therefore run times.
4. **CDFG hierarchy.** We implement hierarchical scheduling by inlining each scheduled subgraph, using a behavioral template to lock the inlined nodes according to the subgraph's schedule.

This paper is organized as follows. Section 2 compares this work to previous research. Section 3 defines behavioral templates. Section 4 describes extending scheduling for behavioral templates. Section 5 discusses applications. Section 6 presents results. Section 7 concludes this paper.

2.0 Related Work

The term "template" was used in [8] to describe structural patterns to exploit regularity. In [9] and [10], such templates are used to guide the clustering of CDFG nodes into super nodes which map to "regular" subcircuits. Both of these works focus on extracting regular patterns by pattern matching, whereas our work focuses on how to schedule a set of behavioral patterns. Our behavioral templates do not represent *repeating* patterns, but specify local scheduling constraints among CDFG nodes.

Most scheduling systems model multi-cycle operations using single CDFG nodes whose delays are greater than 1. In [7], multi-cycle operations are treated as multiple single-cycle operations. This turns out to be similar to our template-based model for sequential operations, except that we make deliberate use of cycle-by-cycle input/output and resource requirements to model complex operations.

Hierarchical scheduling based on super nodes are used in [9], [6], and [7]. We know of no other system which hierarchically schedules a design while taking advantage of possible resource sharing between nodes and edges in different subgraphs.

3.0 Behavioral Templates

We define a behavioral template, T, as a CDFG object which specifies a set of tuples, (n_i, o_i), where n_i is a CDFG node and o_i is an integer cycle *offset*. The semantics is that T imposes the constraint:

$$\text{schedule}(n_i) = \text{schedule}(T) + o_i \qquad \text{for all } (n_i, o_i) \text{ in } T$$

where schedule(n_i) and schedule(T) denotes the schedules for n_i and T, respectively.

That is, if T is scheduled to cstep j, then every member node, n_i, of T must be scheduled to the cstep, $j + o_i$. This locks all nodes in T into a pattern of relative schedules, and we may schedule the entire group of nodes by scheduling the template T itself. Fig. 1(a) shows a template, T1 = { (a,0) (b,1) (c,2) (d,3) (e,5) }, containing 5 nodes. All CDFG edges have been omitted for clarity. (In the figures, we show behavioral template as a box containing one or more nodes in

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX A

slots. The top slot in the box is offset 0, the second slot from top is offset 1, and so on. For example, the node "e" in Fig. 1(a) has offset 5 in T1 because it is in the 6th slot from the top of the box.)

Whenever a node is a member of two or more different templates, we can always merge these templates into one. Consider the templates T1 and T2 in Fig. 1(a) and 1(b). If node g is to be added to template T1 at offset 1, then we merge T1 and T2 into T3 of Fig. 1(c).

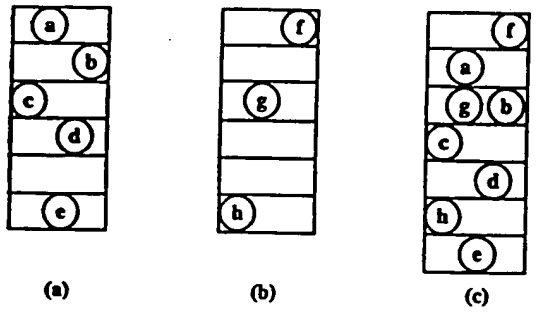


FIGURE 1. Template examples (a) T1 = { (a,0) (b,1) (c,2) (d,3) (e,5) }; (b) T2 = { (f,0) (g,2) (h,5) }; (c) T3 = { (f,0) (a,1) (g,2) (b,2) (c,3) (d,4) (h,5) (e,6) }

4.0 Scheduling with Behavioral Templates

Instead of scheduling individual CDFG nodes, we restate the scheduling problem in terms of behavioral templates. Initially, we create one template for every CDFG node, and then merge templates whenever nodes are added to other templates. This ensures that every CDFG node is a member of one and only one template. The timing constrained scheduling task is then to schedule all templates to minimize resource costs subject to timing constraints between templates. This section describes how we extend existing scheduling algorithms for behavioral templates.

4.1 Timing Constraints between Templates

From the CDFG, we construct a weighted, directed graph G=(V , E) where V is the set of all behavioral templates in the CDFG, and E is the set of directed edges between templates. The weight $d(T_x, T_y)$ of an edge $e(T_x , T_y)$ in E specifies the minimum delay between the schedules of T_x and T_y, i.e.,

$$\text{schedule}(T_x) + d(T_x , T_y) <= \text{schedule}(T_y) \qquad \text{Eq. 1}$$

The edges in E are constructed from the data/control dependencies between member nodes in the templates. For every pair of templates, T_x and T_y, $d(T_x , T_y)$ is the maximum value of

$$w(n_i , n_j) + o_i - o_j \qquad \text{Eq. 2}$$

over all (n_i , o_i) in T_x and all (n_j , o_j) in T_y, where $w(n_i , n_j)$ is the minimum cycle delay from node n_i to node n_j.

Note that Eq. 2 can be negative. This means $d(T_x , T_y)$ can be negative, and the graph G is not acyclic. If G contains any cycle of positive lengths, then the timing constraints are unsatisfiable. To check for positive cycles, we solve for the all-pairs-longest-path problem for G using a simple $O(N^3)$ algorithm, where N is the number of templates in G. The longest path lengths are stored in a matrix, LP, for subsequent incremental update of the as soon as possible (ASAP) and as late as possible (ALAP) schedules.

4.2 ASAP and ALAP Schedules

At the start of scheduling, we calculate the ASAP and ALAP schedules for all templates in G, to establish the scheduling time frame for each template. Since G may contain negative weighted edges, we use a relaxation algorithm similar to that in [3] to compute the initial ASAP/ALAP schedules:

(1) Propagate along positive edges in E only;

- for ASAP, propagate forward from the source of CDFG;
- for ALAP, propagate backward from the sink of CDFG;

(2) Relax schedules to satisfy constraints implied by negative edges in E;

(3) Repeat step 1 until no more changes in relaxation step.

When there are no positive cycles in G, the above algorithm is guaranteed to converge in e+1 iterations where e is the number of negative edges in E. The overall computational complexity is $O(N^2e)$ where N is number of templates in G.

The ASAP and ALAP schedules define the initial time frames. Subsequently, as each template is scheduled, we update the time frames of other templates using the longest path lengths matrix, LP:

$$\text{schedule}(T_x) + LP(T_x , T_y) <= \text{schedule}(T_y) \text{ for all } T_x , T_y \text{ in V}$$

There is no need for relaxation in this incremental update because LP already takes into account all negative edges in E.

4.3 Cost Functions

We use a number of iterative/constructive scheduling algorithms each of which successively picks an unscheduled template and schedules it to a cstep in its time frame. The algorithms differ in how they pick the next template to schedule, and in how they pick which cstep to schedule the template to. We define the template priority/cost functions in terms of priority/cost functions on the CDFG nodes.

For example, in our implementation of list scheduling, the template priority function is defined as the maximum of its member nodes' priority values. This gives priority to the template containing the highest priority nodes. In our implementation of greedy scheduling, the incremental cost function for scheduling a template $T=\{(n_i , o_i)\}$ to a cstep j, is defined as the sum total of the incremental costs for scheduling nodes n_i to csteps $j + o_i$.

Scheduling/de-scheduling moves on templates are implemented as moves on their member nodes. All data structures are updated as CDFG nodes are scheduled/de-scheduled. In particular, resource costs for functional units, registers and interconnects are still computed according to the lifetimes and mutual exclusivity of CDFG nodes and edges. This approach is easy to implement and leverages previous work on scheduling CDFG nodes.

4.4 Pre-assigned Operations

Allowing negative edges in G requires that we extend scheduling algorithms to handle maximum timing constraints. This is complicated by "pre-assigned" operations, i.e., operations that are

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX A
Sheet 3 of 7

assigned t specific resources before scheduling. Examples of pre-assigned operations are memory read/write perati ns for the same RAM. W use a list scheduling algorithm to find an initial legal schedule based on source code ordering. However, list scheduling can fail to find a legal schedule when there are maximum timing constraints. So we augment list scheduling with a recovery step. When list scheduling fails, the recovery step relaxes the template schedules that caused scheduling failures, and iterates:

1. List scheduling step:

Successively consider operations in the ready list in increasing source code ordering. For each ready operation, n_i, check its template, $T_x = \{...(n_i, o_i)...\}$, for scheduling in the cstep s - o_i, where s is the current cstep. Postpone scheduling of T_x if any of the following is true:

- T_x has a "relaxed cstep" (see step 2) which is greater than s - o_i
- T_x has no "relaxed cstep", but ASAP is greater than s - o_i
- there is a resource contention if T_x is scheduled to s - o_i

When all nodes have been scheduled, exit with success.

If T_x is postponed due to resource contention, and if s - o_i is greater than or equal to the ALAP cstep for T_x, then list scheduling has failed. When this happens, go to step 2 and try to recover.

2. Recovery step:

When list scheduling fails to find a legal schedule for T_x, we try to rec ver by increasing its ALAP cstep and rerun list scheduling in step 1. In order to increase the ALAP cstep for T_x, we find all scheduled templates, T_y, for which

$$alap(T_x) == schedule(T_y) - LP(T_x, T_y) \qquad \text{Eq. 3}$$

where $alap(T_x)$ is the ALAP cstep for T_x. For every such template, T_y, we set its "relaxed cstep" to $schedule(T_y) + 1$. This forces the next run of list scheduling to schedule T_y one cstep later.

This step exits with failure if any of the following is true:

- $ALAP(T_x)$ is at maximum global cstep
- there is no template T_y which satisfies Eq. 3
- algorithm has iterated for N times (N is the # of templates)

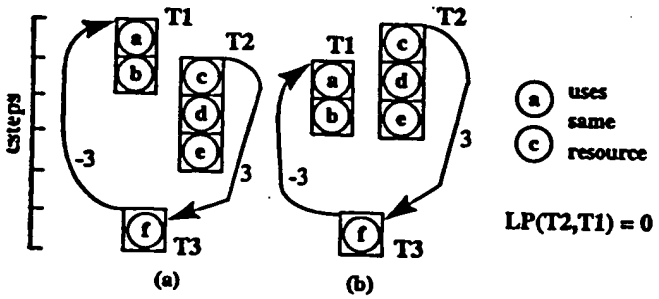


FIGURE 2. Example of list scheduling failure and recovery (a) first iteration fails at T2 (b) second iteration succeeds with T1 relaxed to cstep 1

Fig. 2 shows an example of this algorithm at work. In this example, CDFG nodes "a" and "c" are pre-assigned to the same resource. The source code ordering has "a" before "c" before "f". Initially, list scheduling in step 1 will schedule T1 to cstep 0, and then fails to schedule T2 because of resource contention at cstep 0, and because its ALAP cstep is 0 nce T1 is scheduled to 0. In step 2, T1 will be assigned a relaxed cstep of 1. In the next iteration, list scheduling first schedules T2 t cstep 0, then schedules T1 t cstep 1 t avoid resource contenti n, and finally schedules T3 t cstep 3.

We have two recourses when the above algorithm fails: First, we can continue to try other scheduling algorithms which may still find legal schedules. Second, we can insert precedence constraints t sequentialize pre-assigned operations by their source code ordering. Any unsatisfiable maximum timing constraints would then be detected as positive cycles in the graph G.

5.0 Applications for Behavioral Templates

This section highlights how we use behavioral templates to advantage. Fig. 3 shows the overall flow of our scheduling process.

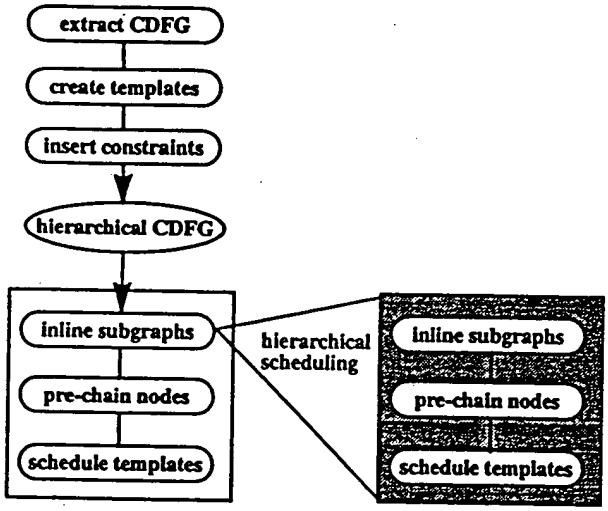


FIGURE 3. Overall flow for hierarchical scheduling

5.1 Inserting Timing Constraints

After extracting the CDFG and creating the initial templates, user-specified timing constraints are added to the CDFG. Minimum timing constraints are represented by precedence edges between nodes, but fixed timing constraints and maximum timing constraints are represented with the help of behavioral templates. Fixed timing constraints are when two or more operations must be scheduled in a fixed number of cycles apart. This is represented by adding one operation to the template of the other operation with the proper offset. For example, if n_j must start k csteps after n_i starts, and if n_i is in template $T = \{...(n_i, o_i)...\}$, then we add n_j to T at offset $o_i + k$ (Fig. 4(a)); if n_j must start k csteps after n_i *ends*, and n_i has a delay of d cycles, then we add n_j to T at offset $o_i + d - 1 + k$ (Fig. 4(b)).

However, if n_j must start k csteps after n_i ends, and n_j does not have a static delay (e.g., n_i is a subgraph), then we decompose the fixed constraint into a k-cycle minimum timing constraint from the end f n_i to the start f n_j, plus a k-cycle maximum timing constraint from the start of n_j to the end of n_i. This is shown in Fig. 4(c).

Note that in Fig. 4(c), we create a dummy *place holder* node, ph, and lock it in a template with n_j (k cycles apart). This template combines with the precedence edge of weight 0 from ph to the end of n_i

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX A
Sheet 4 of 7

to represent the maximum timing constraint from the start of n_j t the end of n_i. The precedence edge f weight k from the end of n_i t start f n_j, represents the minimum timing constraint from the end of n_i to the start f n_j.

In general, a maximum timing constraint f k cycles from a set f nodes, A, to the set of nodes, B, is represented by creating two place holders, ph1 and ph2, fixed k cycles apart in a template, and inserting a 0-weight precedence edge from ph1 to all nodes in A, and inserting a 0-weight precedence edge from all nodes in B to ph2. Fig. 9(a) contains an example of this where the dummy place holder nodes, t2 and t3, are used to lock a write operation to 0 cycle after the end of a loop.

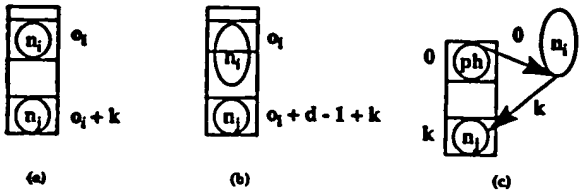


FIGURE 4. Timing constraints (a) n_j starts k cycles after n_i starts, (b) n_j starts k cycles after n_i ends and n_i has static delay d, (c) n_j starts k cycles after n_i ends and n_i's delay is not static.

5.2 Modeling Multi-cycle Operations

Behavioral templates also help model complex multi-cycle operations. When a single CDFG node is used to model a multi-cycle operation, it imposes some limitations due to CDFG semantics:

- Execution cannot start until ALL inputs are available.
- ALL inputs must be held stable throughout operation execution.
- ALL outputs are produced in the last cycle of execution.

This makes it difficult to model, for example, a 3-cycle RAM write operation where the address must be stable for the first two cycles, the data must be stable in the second cycle, and the write sequence finishes in the third cycle. To model such complex operations, we differentiate between combinational and sequential multi-cycle perations. A combinational operation has cycle synchronous inputs and outputs, so it is modeled by a single CDFG node. A sequential operation can have different cycle-by-cycle input/output connections and even resource requirements, and is modeled by several CDFG nodes that are locked into consecutive csteps by a behavioral template. Fig. 5 shows the single-node and multiple-nodes-in-a-template models for the above 3-cycle RAM write operation. Note that in our model (Fig. 5(b)), the address and data inputs are de-coupled in terms of when and for how long each input must be stable. This also de-couples multiple outputs (if any).

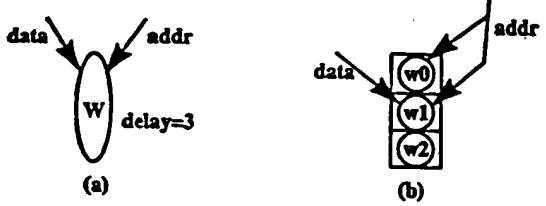


FIGURE 5. Models for a 3-cycle RAM write operation: (a) single node with delay = 3; (b) 3 nodes locked in a template.

This multiple-nodes-in-a-template model is ven more powerful when resource requirements are added. For example, a pipelined peration uses different pipe-stage in different cycles, allowing verlapping pipelined operations to share the same hardware module as l ng as they do not have resource contenti n n any pipe-stage. If we view pipe-stages as internal resources and assign each pipe-stage a named "token", then we may label each node in the template model with the resource tokens it requires. As a node is scheduled to a cstep, we reserve its resource tokens for that cstep. The number of conflicting tokens (i.e., number of non-mutually-exclusive nodes that require the same token) in any cstep gives the number of pipelined modules needed in that cstep. Overlapping of pipelined operations can be scheduled on the same module because successive nodes in the template model require different tokens.

This removes assumptions about pipelined operations from the scheduling algorithms. We may now model operations on complicated pipelines, and template-based scheduling will properly schedule these operations on the pipelined modules. Fig. 6 shows examples of operations on pipelines with internal feedback, sequential inputs, and multiple outputs. We use "a[s1]" to denote an peration named "a" which requires the token "s1".

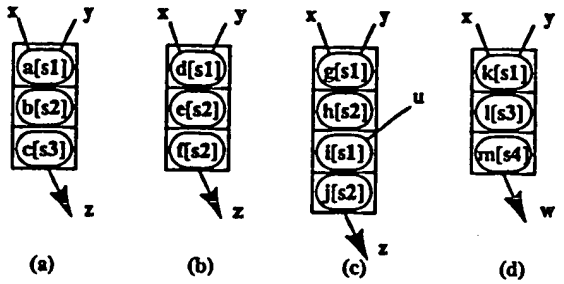


FIGURE 6. Template models for: (a) basic 3-stage pipelined operation, (b) 3-cycle pipelined operation with 2 stages and internal feedback, (c) 4-cycle pipelined operation with 2 stages and sequential inputs, (d) pipelined operation using a different internal path and output port.

Actually, resource tokens need not correspond to physical hardware resources, but may be considered a more general mechanism for specifying how different types of operations can overlap in time on the same module. Consider a 2-cycle RAM which has one read-port and one write-port, whose read/write cycles must be synchronized. Fig. 7 shows how we use resource tokens to specify this constraint to scheduling. If two such operations are pre-assigned to the same RAM, then resource contention on any of "s1", "s2", "s3" and "s4" implies an illegal schedule. The token "s3" prevents read operations for the same RAM to overlap; the token "s4" prevents write perations for the same RAM to overlap; and the tokens "s1" and "s2" prevent read and write operations for the same RAM from being scheduled exactly one cycle apart.

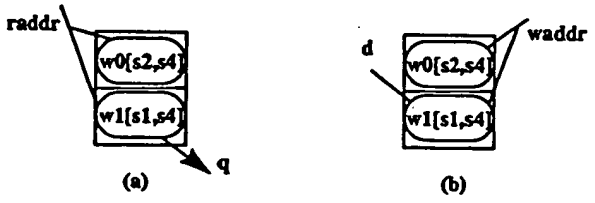


FIGURE 7. Template Models for RAM (a) 2-cycle read, and (b) 2-cycle write.

To handl multi-port RAM's, we allow a module to carry more than 1 copy of a given resource token. For example, to model a 4-port

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX A
Sheet 5 of 7

RAM where each port can be used for both read or write, we would define the RAM module to have 4 "r/w" tokens, and model read and write operation on this RAM to require 1 "r/w" token each. This would allow scheduling to perform up to 4 simultaneous read or write operations on the same module.

5.3 Pre-Chaining

Just before scheduling, we selectively force operation chaining by locking operations in the same cycle using behavioral templates. This "pre-chaining" step reduces scheduling complexity at the expense of scheduling freedom. User-specified chaining directives are applied in this step. We also implement automatic pre-chaining for logic operations and bit-manipulation operations to save registers.

Logic operations include bit-wise AND, OR, NOT, EXOR operations, and bit-manipulation operations include bit-extract, bit-concatenate, constant bit/word generator operations. These operations are good candidates for pre-chaining because they have small propagation delays and they are not resource shared. Thus pre-chaining can be done on the basis of register costs alone. We implement a greedy algorithm for pre-chaining:

1.In a forward traversal of the data flow graph, pre-chain a logic/bit-manipulation operation with its predecessors if there are fewer output bits than input bits;

2. In a reverse traversal of the data flow graph, pre-chain a logic/bit-manipulation operation with its successors if there are fewer input bits than output bits.

3. Iterate until there are no more changes.

Fig. 8 shows examples of good pre-chaining configurations.

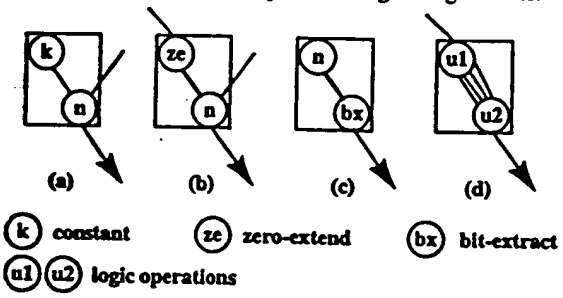


FIGURE 8. Pre-chaining examples: (a) constant with successor, (b) zero-extension with successor, (c) bit-extract with predecessor, (d) multi-input logic with predecessor or multi-output logic with successor

5.4 Hierarchical Scheduling

As shown in Fig. 3, our extracted CDFG is hierarchical, in which each level of the hierarchy corresponds to a loop or a subroutine. Hierarchical scheduling proceeds in a bottom-up traversal of the hierarchy. At each level, instead of representing subgraphs as super nodes, we inline each subgraph and use a behavioral template to interlock the inlined nodes according to the subgraph's schedule.

Inlining subgraphs allows certain boundary optimizations. First, unused subgraph outputs (and the operations that produce these outputs) can be deleted. This deletion can recurs to unused subgraph inputs and then to operations that feed these inputs. Second, inlining subgraphs allows scheduling of neighboring nodes to take advantage of when individual subgraph inputs/outputs are actually required/produced, whereas representing subgraphs as super nodes would force scheduling to assume that all subgraph inputs/outputs are required/produced in the same cycles.

Another advantage of inlining subgraphs is that scheduling maintains accurate cycle-by-cycle resource costs. This allows, for example, to calculate resource costs of scheduling operations in the first and last cycles of a loop subgraph. (When an outside operation is scheduled in the first/last cycle of a loop it is performed when entering/exiting the loop).

In fact, hierarchical scheduling is used to implement sequential multi-cycle operations. In the initial CDFG, each sequential operation is a subroutine call to some library function, which is a pre-scheduled CDFG whose nodes are labelled with the required resource tokens. During inlining, each sequential operation is replaced by an inlined copy of its function's CDFG, and a template is created to lock these inlined nodes. This creates the multiple-nodes-in-a-template model for sequential operations.

The disadvantage of inlining subgraphs is that more nodes are scheduled instead of a small number of super nodes. This is balanced somewhat by the fact that inlined nodes are grouped by templates into a few scheduling units, so at least the scheduling solution space is not much bigger.

6.0 Results

Behavioral templates have been implemented in the Synopsys Behavioral Compiler™ product. Behavioral Compiler™ inputs a VHDL or Verilog behavioral description, performs scheduling, allocation, module selection, binding, and control optimization, and outputs a RTL design which is then optimized by RTL optimization [2], FSM optimization, and logic synthesis.

We will use "dft", a discrete fourier transform design, to illustrate behavioral templates. On reset, dft sequentially reads in the real and imaginary parts of the coefficients into arrays cmem and dmem. These arrays are mapped to the memory "CRAM" (cmem in the lower bank, dmem in the upper bank). It then enters the main processing loop. In each iteration, dft signals it is ready for processing, do a busy wait for the "start" signal, and then sequentially reads in the real and imaginary parts of the data points into arrays amem and bmem, which are also mapped to two halves of a memory, "DRAM". It then enters two nested FOR loops which compute the discrete fourier transform values and write them out. The memories are two-cycle RAM's whose read/write models are shown in Fig. 7. The multiply operations are done on a 2-stage pipelined multiply.

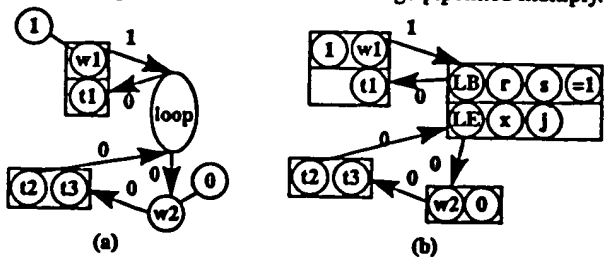


FIGURE 9. Handshaking for start signal: (a) original CDFG with timing constraints, (b) final CDFG scheduled

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX A
Sheet 6 of 7

Fig. 9 sh ws the CDFG fragment for the busy-wait on the "start" signal. Fig. 9(a) shows the initial CDFG containing the fixed constraints that the "ready" signal be asserted one cycle before the busy wait, and deasserted 0 cycle after the busy wait. Fig. 9(b) shows the sam CDFG fragment that is finally scheduled. By this time, hierarchical scheduling has already scheduled the busy wait loop, and the loop body is inlined and locked in a template. Also, pre-chaining has locked the constants with the write operations.

However, the main scheduling problem is in the inner-most loop, which reads from memories the complex data *(a , jb)*, and coefficient *(c , jd)*, and computes *psum* += *(a*c - b*d)* and *ipsum* += *(a*d + b*c)*. Table 1 shows the scheduling and allocation results for the computation part of this loop. Note the pipelined RAM read's are chained with the pipelined multiply operations.

amem	bmem	cmem	dmem	p-mult		+/-
r0[s1]			r6[s1]			
r1[s2]	r2[s1]	r4[s1]	r7[s2]	x0[s1]		
	r3[s2]	r5[s2]		x1[s2]	x2[s1]	
				x4[s1]	x3[s2]	add1
				x5[s2]	x6[s1]	add2
					x7[s2]	sub1
						add3

TABLE 1. Scheduling/allocation results for dft's inner loop

FIGURE 10. Scheduling/allocation result for computation part of dft's inner-most loop

In Table 2 and 3, we present some design statistics. #line is number of VHDL/verilog lines in the source. #loop is number of loops with nesting levels in brackets. #node is number of CDFG nodes. #template is total number of templates scheduled. The ratio of nodes to templates are shown in brackets. #RAM is the number of on-chip memories used. #gate is gate count after logic synthesis (excluding RAM's). Note the difference between #node and #templates.

Table 2 presents several HLSW benchmarks, modified to use more realistic bit-widths. EWF is the fifth order elliptic wave filter example (20 csteps for the main loop, using 1 16x16 pipelined multiply, 2 32-bit adders, 10 32-bit registers and 1 16-bit register). KF is the Kalman filter modified to use 5 RAM's.

	#line	#loop	#node	#template	#RAM	#gate
EWF	128	2 (2)	103	78 (1.32)	0	9677
KF	207	10 (4)	261	144 (1.81)	5	7963
i8251	596	54 (3)	1238	625 (1.98)	0	5274
gcd	57	2 (2)	77	22 (3.50)	0	825

TABLE 2. Design statistics for several HLSW benchmarks

	#line	#loop	#node	#template	#RAM	#gate
dft	218	5(3)	151	65 (2.32)	2	3920
rgb filter	247	3 (2)	286	109 (2.62)	4	4316
idct	274	8 (3)	902	287 (3.14)	3	32677
viterbi	262	3 (2)	1797	296 (6.07)	0	3653
graphics ctrl	123	2 (2)	98	30 (3.27)	0	4071
high pass	389	5 (2)	310	153 (2.03)	11	8970

TABLE 3. Design statistics for several industrial examples

Table 3 lists several industrial xamples. Compared to benchmark examples, these designs tend to:

- have more complicated reset sequences before the main processing loop,
- have more cycle-by-cycle timing constraints n IO operations,
- have more logic operations and bit-manipulation operations,
- use multiple RAM's r multi-port RAM's t improve RAM access bottlenecks,
- use pipelined operations t increase throughput.

7.0 Conclusion

In this paper, we have presented our work on scheduling using behavioral templates. The most important value of behavioral templates is that they enable simple solutions to the problems of (1) enforcing fixed and maximum timing constraints, (2) modeling complex sequential operations, (3) pre-chaining of logic and bit-manipulation operations, and (4) hierarchical scheduling. For this reason, behavioral templates have been instrumental in our productization of behavioral synthesis.

Future work will investigate adding structural templates to partition the design based on structural regularity.

8.0 Acknowledgment

We would like to acknowledge Russ Segal and Dennis Fogg, who designed and implemented library interface for our sequential operations. We would also like to acknowledge Pradeep Fernandes and Hazem Almusa for helping us collect the results reported in Tables 1, 2 and 3.

9.0 References

[1] R. Camposano, "Path-based scheduling for synthesis", IEEE transactions on Computer-Aided Design, vol. CAD-10, pp 85-93, Jan 1991.

[2] B. Gregory, D. MacMillen & D. Fogg, "ISIS: A System for Performance Driven Resource Sharing", in Proc. of 29th DAC, pp285-290, June 1992.

[3] D. Ku & G. De Micheli, "Relative Scheduling under Timing Constraints", in Proc. of 27th DAC, pp59-64, June, 1990.

[4] M.C. McFarland, A. C. Parker & R. Camposano, "The High-Level Synthesis of Digital Systems", in Proc. of IEEE, vol. 78, pp301-318, Feb. 1990.

[5] P. Michel, U. Lauther and P. Duzy, "The Synthesis Approach to Digital System Design", Kluwer Academic Publishers, 1992.

[6] S. Note, W. Geurts, F. Catthoor & H. De Man, "Cathedral III: Architecture-Driven High-Level Synthesis for High Throughput DSP Applications", in Proc. 28th DAC, pp597-602, June, 1991.

[7] M. Nourani & C. Papachristou, "Move Frame Scheduling and Mixed Scheduling-Allocation for the Automated Synthesis of Digital Systems", in Proc. 29th DAC, pp99-105, June, 1992.

[8] D. S. Rao & F. J. Kurdahi, "Partitioning by Regularity Extraction", in Proc. 29th DAC, pp235-238, June, 1992.

[9] D. S. Rao & F. J. Kurdahi, "System Modeling for High-Level Synthesis Using Regularity Extraction", D. S. Rao & F. J. Kurdahi, in Proc. Sixth International Workshop on High Level Synthesis, pp267-272, Nov 1992.

[10]K. Rose & C. T. Chang, "Cluster-Oriented Scheduling in Pipelined Data Path Synthesis", in Proc. ICCD, Oct. 1993.

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX A

Appendix B

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street.
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX B
Sheet 1 of 7

Behavioral Synthesis Methodology for HDL-Based Specification and Validation

D. Knapp, T. Ly, D. MacMillen, R. Miller

Synopsys Inc.
700B E. Middlefield Rd
Mountain View, CA USA 94043

Abstract

This paper describes a HDL synthesis based design methodology that supports user adoption of behavioral-level synthesis into normal design practices. The use of these techniques increases understanding of the HDL descriptions before synthesis, and makes the comparison of pre- and post-synthesis design behavior through simulation much more direct. This increases user confidence that the specification does what the user wants, i.e. that the synthesized design matches the specification in the ways that are important to the user. At the same time, the methodology gives the user a powerful set of tools to specify complex interface timing, while preserving a user's ability to delegate decision-making authority to software in those cases where the user does not wish to restrict the options available to the synthesis algorithms.

1.0 Overview

This paper describes a synthesis methodology that uses high-level synthesis (HLS) of behavioral hardware-description language (HDL) descriptions. HLS has the distinguishing characteristic that operations are automatically *scheduled*, i.e. assigned to states, as opposed to lower-level synthesis, in which operations are assigned to states by the user [1, 2, 3]. For example, in an HDL description of a square root function, an operand x would be loaded, a series of operations would foll w, and a single result r would be returned. The read x and the write r might be fixed to particular states or times by a communication protocol, but the internal operations that compute the square root would be automatically scheduled.

A prospective user of HLS will then ask a number of questions. These will likely include the following:

- How can I constrain I/O operations to fall into particular cycles, or range of cycles, to meet existing protocols?
- How can I constrain I/O operations to have particular timing relationships? For example, how can I constrain a data ready strobe to be synchronous with data on data ports?
- How can I be confident that my interface timing specification really works with the surrounding hardware?
- H w can I give the scheduling software optimization opportunities when my timing specification is not rigid? For example, I might not care exactly when data was transferred, as long as a corresponding strobe remains synchronized with the data. Thus the strobe and data should be locked together, but the locked strobe/data pair of operations could move.
- How can I be confident that the synthesized hardware will really d what I want it to:
  1. In the sense that it computes the right result,
  2. In the sense that scheduling of I/O operations does not 'break' its I/O protocols.

These questions can be reformulated as requirements on the HDL description methodology to be used in conjunction with HLS:

- The original HDL description should be simulatable.
- There should be a mode wherein the cycle by cycle I/O timing of the original HDL description is preserved exactly; i.e., no I/O timing difference will be allowed between the pre-and post-synthesis descriptions. This will allow direct comparison, on a cycle by cycle basis, of the pre- and post-synthesis designs; it will also allow the user to meet the most rigid cycle-based timing protocols.
- There should be a mode wherein timing relationships between I/O signals can be simply and easily preserved across synthesis, but where 'stretching' (cycle level delay insertion) is permitted, so that the user does not have to specify exactly how many cycles a computation will take. This mode should allow manual constraints. Such a mode allows comparison of pre- and post-synthesis I/O timing between "similar points" of the pre- and post-synthesis waveforms.
- There should be a mode in which the user explicitly specifies all timing constraints without reference to the simulation behavior of the HDL; the only timing constraints inferred from the HDL description are ordering constraints among I/O operations sharing a port. This mode gives the greatest flexibility, both for optimization and for specification of complex timing relationships; it is also the most difficult to use.

We call these three modes the *cycle-fixed* IO *scheduling mode*, the *superstate-fixed IO scheduling mode*, and the *free-floating IO scheduling mode* respectively. Each has consequences for the style of HDL description and validation methodology. These modes give the user a wide range of choices in specifying I/O timing, with a corresponding range of ways in which validation of the specification and comparison of the implementation with the specification can be performed.

1.1 Structure of this paper

The balance of this paper is structured as follows. In Section 1.2, related work in this field is discussed. Following that, in Section 2, some mode-independent considerations and assumptions are described. In Section 3, the cycle-fixed mode is described in detail. Then in Section 4, the superstate-fixed mode is described. In Section 5, the free-floating mode is described. In Section 6, experience with the current software is described; finally, in Section 7 the paper is summarized and conclusions are drawn.

1.2 Related Work

High-l vel synthesis has been w ll described in th literature; see, f r example, Camposano[1], Gajski[2], Maerz[3]. These tutorial papers describe the basics of HLS systems. CALLAS [4] describes work in the area of maintaining simulated behavior that is exactly the same pre- and post-synthesis; this idea is reflected

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX B
Sheet 2 of 7

in the cycle-fixed mode described here. The superstate-fixed mode is related the High Level State machine of of [5], and t the behavioral finit state machines (BFSM's) of [6]. Our approach of validation through simulation is typical f current industry practice; it complements, but cannot completely replace, more formal methods [7].

2.0 Basic assumptions

The circuit to be synthesized by HLS consists of a collection of always blocks (VHDL processes); each always block will be mapped to hardware consisting of a datapath and a control FSM. Each will be synthesized separately.

Control over timing makes use of clocking statements in the source HDL. In Verilog, this can be done by use of @(posedge clock) or @(negedge clock) statements[1]. These are used to separate I/O vents that are to happen in different clock cycles. Event triggers using other signals are specifically disallowed, with the exception of asynchronous reset and a special gating methodology described in Section 2.2, used for synchronizing I/O.

2.1 Reset

In order to handle resets in an intuitively appealing way, we call attention to the always block (VHDL process) that will be scheduled. In our methodology this block contains a single all-encompassing, nonterminating loop, here called *reset_loop*.

```
always begin: b1
    begin: reset_loop
        // reset sequence behaviors
        forever begin
            // normal mode behaviors
        end
    end
end
```

Inside *reset_loop* is a *reset sequence*; this consists of all behaviors associated with reset. For example, in a microprocessor the reset sequence would clear the program counter, disable interrupts, and initialize the stack pointer. The reset sequence may contain many clock cycles, e.g. to initialize a RAM. Following the reset behavior is the 'normal mode' loop, which does not terminate either; this loop contains behaviors that are executed until the next reset occurs. In a microprocessor, for example, the normal mode loop would be the fetch / execute cycle.

In order to simulate the effect of synchronous resets correctly in the source HDL description, the user must insert a statement of the form[2]

```
if (reset == 1'b1) disable reset_loop;
```

after every @posedge statement. This disable has the effect of restarting the block (process) following a clock edge upon which reset is found to be true. Simulation of synchronous resets can be matched both pre- and post-synthesis.

Another capability can also be provided in which the user declares a reset pin to the synthesis software, which then synthesizes the reset; but because the reset behavior is not encoded in the HDL, resets cannot be simulated correctly before synthesis using this technique.

Scheduling cannot handle exits triggered by a reset in the same way as other exits, because there may be read-bef re-write accesses in the HDL. Consider the foll wing: In this situation, the assignments

```
begin: reset_loop
    outport <= x; // x is read before write!
    begin: main_loop
        x = v1;
        @(posedge clock);
        if (reset == 1'b1) disable reset_loop;
        x = v2;
    end
end
```

of *x* cannot be rescheduled, because this would change the observable behavior of the circuit immediately following a reset pulse. If, for example, the second write to *x* was rescheduled before the clock edge, then the output immediately following a reset pulse would be *v2* in the scheduled design; but it would be *v1* in the original description. So if we are to allow read before write in the HDL, w must either relax the requirement that all behaviors must be identical, or we must forbid movement of such side effects across clock boundaries. Side effects on variables that are always written before they are read are not affected.

2.2 Registered outputs

VHDL signals and Verilog reg variables behave like register or latch outputs. That is, they hold their values once set. For implementation reasons, we chose to register all outputs of HLS synthesized designs; thus a nonblocking (signal) assignment becomes a register write. This has the consequence that responses to external events cannot happen until the cycle after the external event, as shown in Fig. 1.

Figure 1 shows the behavior of a synthesized circuit where the HDL input is of the general form

```
if (Ready == 1'b1) then Data <= foo;
@(posedge clock);
```

This timing corresponds to both input and output. Notice that this

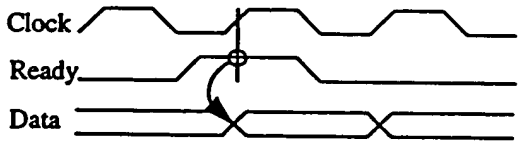


Fig. 1. Response to an external event.

timing diagram implies that the control FSM for the synthesized data path is a Mealy machine; and that the overall synthesized design is a Moore machine.

Here is an example combining an asynchronous reset and a compact busy wait on a data strobe.

```
while (strobe != 1) begin
    @(posedge clock or posedge reset);
    if (reset == 1'b1) disable reset_loop;
end
```

3.0 Cycle-fixed mode

High-level synthesis in *cycle-fixed* mode can be described by the following statement:

- Cycle-by-cycle I/O timing is identical between the pre-and post-synthesis designs.

This means that validation by simulation is straightforward: a user need merely simulate the pre- and post-synthesis designs side by side, and check for differences in the outputs. Alternatively, the synthesized design can be inserted into the original test bench without modifying the test bench. The only differences that are visible involve combinational delays in the form of setup and hold times; for example, a delta-delay setup time w uld become a real setup

1. In VHDL "wait until clock'event and clock = '1';" gives us a rising-edge clock.

2. In VHDL this would be "when reset = '1' exit reset_loop".

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX B
Sheet 3 of 7

time, and a registered output pin will not transition exactly on the clock edge, as it would in the pre-synthesis simulation[1]. This is shown in Fig. 2.

Notice that this mode only constrains the I/O operations of the design. That is, the reads and nonblocking (signal) writes of the HDL are tied to particular cycles. But this still leaves optimization opportunities for the scheduling algorithm: other operations (e.g. additions, memory operations, and register reads and writes) can be shifted in time, as long as they consume data after it has been read in, and produce data in time to write it out. The I/O operations provide a series of 'stakes in the ground' that define time frames within which all other operations are free to move.

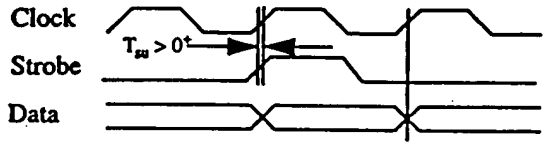


Fig. 2a. Simulation of specified design (pre-synthesis)

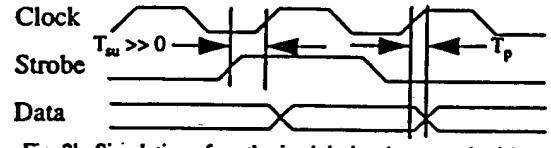


Fig. 2b. Simulation of synthesized design (post-synthesis)

Fig. 2. Comparison of simulation in cycle-fixed mode.

The main advantage of cycle-fixed mode is that the user can synthesize exactly the same timing diagram that the original HDL specification shows in simulation; thus, if the simulated HDL specification works in a particular context, then the synthesized design will also work, assuming only that setup, hold, and propagation delays, etc. as shown in Fig. 1b meet the clock cycle time.

A further advantage of cycle-fixed mode is that simulation of a zero-gate-delay model of the synthesized design will match the original specification exactly; hence a simple file difference program can be used to compare pre- and post-synthesis designs. This is expected to have a profound effect on user acceptance of HLS as a viable tool in the design cycle: users are able to simply and efficiently check the equivalence of designs before and after synthesis.

There are a number of methodological and implementation considerations that affect the way we can write and implement cycle-fixed mode. These will now be described.

3.1 Numbers of clock edges

One consequence of the commitment to maintain exact I/O equivalence in cycle-fixed mode is that numbers of clock edges cannot be varied inside the scope of loops and conditionals. To do so would distort the I/O timing of the design.

3.2 Loop boundaries

Every loop of an always block must contain at least one clock edge statement. The only exception to this is loops with constant iteration bounds, which can be unrolled during synthesis.

A loop can be thought of as a subgraph of a finite-state machine (FSM) which forms a cycle. The synthesized design will enter this cycle when the loop is executed, and leave it when the loop is exited. Such a loop is shown in Fig. 3.

```
o1 <= v1;
while (c) begin: loop
    o2 <= v2;
    @(posedge clock);
end
o3 <= v3;
@(posedge clock);
```

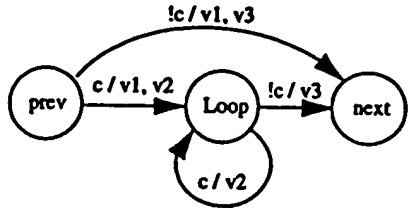


Fig. 3. Loop and corresponding state graph

The loop of Fig. 3 corresponds to the state labeled 'Loop'. During each pass of the loop, the value of *v2* will be written to the output port *o*.

The main consequence of matching this behavior is the splitting of the conditional test *c*. Notice that it was necessary, in order t capture the timing of the original, to have a state transition that bypassed the loop altogether if *c* was false when it was first tested. This means that the test must be performed in two places: once in state *prev*, and once in state *loop*. In general, it is necessary to unroll the first state of the first pass through a while loop in order to capture this behavior correctly.

If we wish to avoid unrolling the first pass, then it is necessary to rewrite the loop so that (1) there is a clock edge on all paths between the writes of *o1* and *o3*, and (2) there is a clock edge between the conditional test and any succeeding I/O, as shown in Fig. 4.

```
o1 <= v1;
while (c) begin: loop
    @(posedge clock);
    o2 <= v2;
end
@(posedge clock);
o3 <= v3;
```

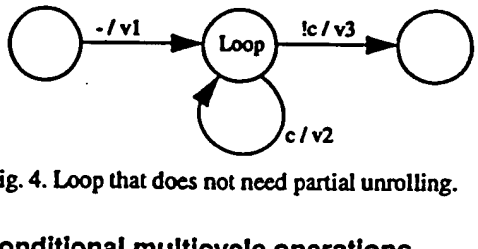


Fig. 4. Loop that does not need partial unrolling.

3.3 Conditional multicycle operations

A *multicycle* operation is one that has a longer combinational delay than the clock cycle. This imposes special constraints n synthesis in cycle-fixed mode, because it is necessary to stabilize all data and control inputs to the hardware block that implements the multicycle operation. This includes all the control inputs of all multiplexers

1. In zero-delay simulation one should ensure that data transitions occur slightly after clock transitions; failing to do this is the most common source of simulation mismatches. The problem comes about because of varying numbers of simulation-cycle delays in th clock and data wires of the circuit: the clock can arrive 'after' the data by an infinitesimal (zero-time) amount. This causes something analogous to a setup-time violation.

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX B
Sheet 4 of 7

that drive multicycle operations; clearly we cannot afford glitches n these paths.
But inserting these registers means that we need to know what t strobe into the registers one cycle before the multicycle operati n is to begin. Thus we need to add extra time, under some circumstances, so that the stabilizing registers can be properly loaded. This is illustrated in Fig. 5; we assume

```
@(posedge clock);
if (input_signal == 1'b1) begin
        x = input_read_1;
        y = input_read_2;
        tmp = x + y; // 2 cycle addition
        @(posedge clock); // strobe stab regs
        @(posedge clock); // 1st cycle of add
        @(posedge clock); // 2nd cycle of add
        out <= tmp;
end
@(posedge clock);
```

Fig. 5. HDL description for a multicycle addition.

Notice that we needed three clock cycles to do this properly: one to get the condition and strobe the stabilizing registers, and two to perform the multicycle addition. Notice also that such delays can ften be hidden, where the multicycle operations are not constrained by I/O; but that in this case there is no opportunity to hide the additional delay associated with stabilizing the inputs.

3.4 Loop pipelining in cycle-fixed mode

Loop pipelining is a technique whereby a loop can be made to act like a pipeline. Thus the loop has a relatively long latency, i.e. the time from a data input to the corresponding data output; and a sh rter initiation interval, which is the rate at which data can be delivered to and read out from the loop. In cycle-fixed mode, and with some extra constraints in the other modes, a simple way to imply loop pipelining while maintaining timing equivalence is to use a delayed assignment (in VHDL, a transport delay) on the output statement. Suppose, for example, we have a loop whose latency is ten cycles, but whose initiation interval is two cycles; we can put an output write after the second clock edge statement, with a delay of eight cycles. This will simulate the same way both before and after synthesis.

```
while (condition) begin
    @(posedge clock); // 10 ns clock
    @(posedge clock);
    out <= #80 value; // delayed by 8 cycles
end
```

4.0 Superstate-fixed Mode

The *superstate-fixed* I/O mode is used where the I/O should inherit its general structure from the HDL, but where there is some freedom to shift I/O operations in time. Consider, for example, the two-wire handshaking protocol shown in Fig. 6.
The two-wire protocol is insensitive to the time between transitions; this makes it ideal for many applications. In a case like this, the only things we really need to assure in order to have correct timing are that (1) the signal transitions occur in the right order, and (2) that the transitions of *Strobe* and *Data* maintain a lockstep relationship. Beyond that, the user might not care very much how many clock cycles were inserted by scheduling; other design optimization criteria (such as the number of gates to compute the data value) might dictate more or fewer clock cycles for this transaction. The cycle-fixed mode is unsuitable f r this kind of loosened specification of timing: the user could be f rced to edit the code many times, with varying numbers of clock edge statements each time, looking for the best implementation.

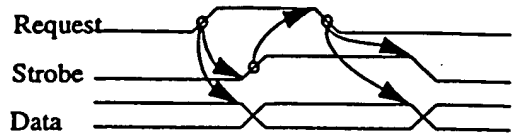


Fig. 6. Two-wire handshaking protocol.

The *superstate-fixed I/O scheduling mode* can be expressed by the following statements:

- Adjacent pairs of clock edge statements in the HDL form the boundaries of superstates.
- All I/O operations in a superstate remain in that superstate.
- A superstate may be expanded by the scheduler, which can add clock cycles to lengthen a superstate.
- All I/O writes in a superstate will always take place in the last clock cycle of the superstate.
- I/O reads may float within a superstate.

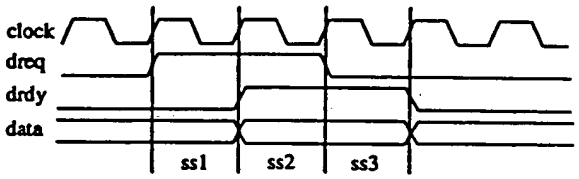


Fig. 7a. Simulation before superstate-fixed scheduling.

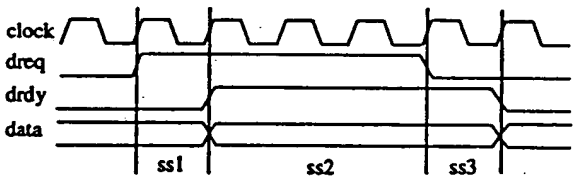


Fig. 7b. Simulation after superstate-fixed scheduling.

These rules, taken together, mean that an HDL scheduled in superstate mode will show the same signal transitions and ordering as the original HDL; but that the original timing may potentially be 'stretched' by the addition of new clock edges. This is illustrated in Fig. 7, where the original HDL simulation of an I/O transfer taking three cycles has become five cycles long by the addition of two extra cycles to the second superstate.

4.1 Protocols in superstate mode

One of the major advantages of superstate mode is that handshaking I/O protocols are not distorted by the addition of clock cycles to superstates. This has two beneficial conseqences: first, comparison of simulated pre- and post-synthesis designs is straightforward; and second, protocols that are insensitive to increased numbers of clock cycles will not be 'broken' by superstate scheduling. Hence if a design consists of many processes, each of which is to be scheduled, the use of handshaking communication in conjunction with superstate mode scheduling will ensure that the design will continue to work after synthesis. The same considerations apply to the simulation test bench as well: the test bench must communicate with the synthesized design(s) via handshaking protocols; otherwise it may have to be modified to communicate successfully with the synthesized design. This happens because the read and write operations occur at different times pre- and post-synthesis; the test bench must be able to tolerate this, or the user will have to retime the test bench.
Protocols that do not involve explicit requests and acknowledges can still be used; but care must be taken with data to be read in by the syn-

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX B
Sheet 5 of 7

thesized process. In particular, recall that read operations may move freely within their superstate. This means that data being presented to the synthesized circuit must be either valid during the entire superstate in which it is read, or else retimed after scheduling. This will ensure that the read perati n always gets the correct data.

4.2 Constraints in superstate mod

The reason a designer would use superstate mode instead of cycle-fixed mode is that some part of the schedule does not have a fixed timing bound, and the user does not want to imply such a bound by using cycle-fixed I/O. However, the user may have a non-handshaking protocol, or a protocol that streams data once synchronization has been established by the protocol. In such cases the parts of the schedule that perform synchronization may need to be handled as if the scheduler was in cycle-fixed mode; while the other parts of the design can be allowed more freedom. For example, consider the fragment

```
while (ready == 1'b0) begin: handshaking_loop
    @(posedge clock);
end
@(posedge clock);
a1 = in_port; // label read_1
@(posedge clock);
a2 = in_port; // label read_2
@(posedge clock);
out_port <= long_involved_function(a1, a2);
out_ready <= 1'b1; // label done
@(posedge clock);
```

Here the external logic provides the data for *read_1* and *read_2* in the two cycles after the signal *ready* goes true; the synthesized system must pick it up then, or the protocol will be broken. Furthermore, insertion of extra cycles in the loop *handshaking_loop* will cause the interface to behave unpredictably. Thus cycle-fixed mode would seem to be indicated. However, suppose that there is no need for the output to show up until 20 cycles after the input has been delivered; the designer will thus want to allow the scheduler authority to add cycles to the last superstate, and rely on a test of the *out_ready* pin to synchronize the data on *out_port*. Thus stretching can be allowed in the last superstate, but not in the first three.

This can be done by means of explicit point-to-point scheduling constraints; that is, constraints that tie two labeled operations together in a particular timing relationship. A constraint set that would serve the purpose is

1. The time from the beginning of *handshaking_loop* to its end should be exactly one cycle.
2. The time from the end of *handshaking_loop* to the beginning of *read_2* should be exactly one cycle.
3. The time from the end of *handshaking_loop* to the data ready strobe *done* is no greater than 21 cycles.

N tice that these constraints are not part of the HDL; but they are a necessary part of the methodology. They can be implemented as pseudo-comments, as attributes, or as directives in a separate scheduler command file. Notice also that they can be applied to non-I/O operations as well, in all three modes, to give the user a little extra control over the scheduling process.

4.3 Superstate HDL methodology

Superstate mode defines superstates as containing the I/O operati ns that fall between adjacent pairs of clock edge statements. This definition has the consequence that sometimes an HDL prepared f r superstate mode needs clock edge statements that are not needed in cycle-fixed mode. For example, the text f Fig. 3 is ambiguous when the HDL is considered as input for superstate mode. This comes about because tw writes are separated by a conditional @posedge. If the loop condition is true, then the writes should be in different superstates; if it is false, then they should be in the same superstate. Clearly there is n unique static assignment of I/O perations to superstates in this situati n.

Furthermore, there is an implicit rdering of operations conferred by the sequencing of the HDL text; this ordering cannot be all wed to come into conflict with the ordering conferred by the migration of reads into any cycle of their superstate and writes into the last cycle of their superstate.

The HDL methodology rules that prevent ambiguities and contradictions in superstate mode are:

1. A superstate that contains a loop continue is called a continuing superstate. Implicitly, the last superstate of a loop is also a continuing superstate. A continuing superstate and the first superstate of the loop are really the same superstate; there is no clock statement on the execution path going from one to the other. If a continuing superstate contains a write, then the first state f the loop cannot contain any I/O, because a write belonging to the continuing superstate would be migrated to the end of the first loop superstate: this would result in a violation of the HDL's ordering constraints.
2. A superstate that contains a loop beginning cannot include both an I/O write before the loop beginning and any I/O operation inside the loop. For example,

```
@(posedge clock);
out_port <= write1_data;
while (cond) begin
    read1_data = in_port; // Illegal!
    @(posedge clock);
    ...
end
```

   the write in this fragment conflicts with the read in the beginning of the loop; they are in the same superstate.
3. A write cannot precede a while loop that is succeeded by any I/O operation, unless there is a clock edge statement between either the write and the loop begin, or between the loop end and the second I/O operation.
4. A loop having a superstate in which both a loop exit[1] and an I/O write are located must have a clock edge statement between the loop end and the next I/O operation.
5. A conditional clock edge (e.g. an @edge on one branch of a conditional) cannot be used to separate a write from another I/O operation. This fragment is illegal for that reason.

```
out_port <= v1;
if (cond) @(posedge clock);
v2 = in_port;
```

5.0 Free-floating I/O mode

It will sometimes be the case that a user will need to convey more freedom to the scheduler than is allowed by the superstate I/O mode. For example, the user may wish to allow two unrelated writes to be permuted. Consider the fragment of Fig. 8.

In this situation, the user might not care whether the first or the second function happens first; indeed, they could be interleaved and the user might not care. But neither superstate nor cycle-fixed mode will permit permutation of I/O operations and waits; so a more powerful mode is needed.

---

1. Other than a reset exit. Reset exits can be ignored after a preprocessing step in which they are detected and global reset behavior is enacted, as explained in Section 2.

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX B

The *free-floating* mode is characterized by implicit constraints on

```
a1 = in_port1;
a2 = in_port2;
@(posedge clock);
out_port_1 <= long_function_1 ( a1, a2 );
@(posedge clock);
b1 = in_port3;
b2 = in_port4;
@(posedge clock);
out_port_2 <= long_function_2 ( b1, b2 );
```

Fig. 8. Writes to out_port_1 and out_port_2 may be permuted.

single I/O ports and explicit user constraints.

Implicit I/O port constraints are derived directly from the HDL text and are imposed on the sets of reads and writes that occur on a single port. These are formed into partially ordered sets, one for each port, where the ordering is derived from a static execution trace analysis of the source HDL. The schedule constructed by synthesis can only transpose two members of one of these sets if there is no ordering relationship between them.

This, however, says nothing about ordering of reads and writes that occur on different ports, which must be explicitly constrained by the user, by means of the explicit two-point constraints described in Section 4.2.

For example, in our experience a common early mistake in free-floating mode is to expect a data strobe's timing to be fixed with respect to that of the data being strobed. This will not necessarily be the case if the user does not issue explicit constraints.

The downside of this mode is the number of explicit constraints that the user must construct. This can easily be comparable in numbers of lines to the HDL input itself. In addition, it is very easy to get such constraints wrong, or to forget a crucial constraint; hence the cycle-fixed and superstate modes are simpler and less error-prone to use.

6.0 Experience

Support for the methodologies discussed above has been built into a commercial product, the Synopsys Behavioral Compiler(TM). This product is currently in use at a number of sites. Of these, about half use Verilog as their input HDL; the rest use VHDL.

Experience to date indicates that the superstate mode is usually the most convenient from the standpoint of ease of specification of complex timing behaviors. The next most convenient is usually the cycle-fixed mode. The reason for this is that the power of the free-floating mode comes at the price of manually added constraints; while the cycle-fixed mode requires the user to add clock cycles to the source HDL when, e.g., the duration of a particular loop is to be changed.

From the standpoint of ease of validation of results, the cycle-fixed mode is usually a little more convenient than the superstate mode. This is because the handshaking protocols necessary to get the design talking to the test bench after superstate-mode scheduling must be designed and written in both the test bench and the specification; or alternatively the test bench timing must be modified to match the schedule of I/O of the post-synthesis design.

One area in which the free-floating mode seems to be more convenient than the others is in that of exploration. Here the user is more interested in getting a rough idea of the cost and speed of a design algorithm, than in getting its interfaces exactly right. In this context, the ease of turning the design around and the high degree of freedom from methodological constraints makes it simpler to change the design and resynthesize to see what the overall results are. Then when the general outlines of the algorithms, representations, etc. are clear the user can begin to worry about the detailed I/O timing.

The overall effort of getting I/O interfaces right using these three modes is usually less than the effort spent in getting the best possible quality of results. Even with behavioral synthesis, HDL writing styles still can have a large impact on the quality of the synthesized circuit. Examples that can affect synthesis quality are: loop ordering, assignment of variables and arrays to memories, choice of loop pipeline initiation intervals and latencies, pipelined components, embedding combinational logic in reusable function blocks, the tradeoff between multicycle operations and fast clock rates, and the partitioning of the design into datapath/controller subunits (i.e. always blocks; in VHDL, processes). All are potentially of great importance to the quality of results, and all represent true engineering decisions that must be carefully considered if a really good design is to be achieved.

7.0 Conclusion

We have presented HDL methodologies for the synthesis of various kinds of I/O timing and protocols, and for simulation-based validation of the synthesized design against the original specification. Three modes of scheduling I/O operations have been presented:

1. Cycle-fixed, in which the design has exactly the same cycle-level I/O timing before and after synthesis;
2. Superstate-fixed, in which I/O operations are grouped by pairs of @posedge statements; post-synthesis timing behavior is a (potentially) stretched version of the pre-synthesis timing; and
3. Free-floating, in which the only constraints on I/O scheduling are either between operations sharing a port or supplied by the user.

Some of the implications of the scheduling modes were described. In the cycle-fixed and superstate modes, these involve the placement of clock edge statements, loop boundaries, conditionals, and I/O operations; while in the free-floating mode there are no rules of this kind.

Experience with production software which implements these methodologies has been described, and conclusions based on that experience have been drawn.

8.0 References

1. R. Camposano, W. Wolf. Trends in High-Level Synthesis. Kluwer, 1991.
2. D. Gajski, N. Dutt, A. Wu, S. Lin. High-Level Synthesis: Introduction to Chip and System Design. Kluwer, 1992.
3. S. Maerz, High Level Synthesis. In *The Synthesis Approach to Digital System Design*, P. Michel, U. Lauther, P. Duzy, eds., Chapter 6. Kluwer, 1992.
4. A. Stoll and P. Duzy, High-Level Synthesis from VHDL with Exact Timing Constraints, *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pp. 188-193, IEEE, 1992.
5. R. A. Bergamaschi, A Kuehlmann, S-M. Wu, V. Venkataraman, D. Reischauer, and D Neumann, A Methodology for Production Use of High Level Synthesis, Workshop Proceedings, Sixth International Workshop on High Level Synthesis, (1992).
6. W. Wolf, S. Takach, C.-Y. Huang, R. Manno, E. Wu. The Princeton University Behavioral Synthesis System. *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pp. 182-187, IEEE, 1992.
7. K. L. McMillan, Fitting Formal Methods into the Design Cycle, *Proceedings of the 31st ACM/IEEE Design Automation Conference*, pp. 314-319, IEEE, 1994.

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX B
Sheet 7 of 7

EXHIBIT 2

THIS PAGE BLANK (USPTO)

EXHIBIT 2

TO

DECLARATION OF JONATHAN T. KAPLAN AND STATEMENT OF FACTS IN SUPPORT OF FILING ON BEHALF OF NON-SIGNING INVENTOR

Pursuant to 37 CFR 1.47

THIS PAGE BLANK (USPTO)

FedEx Express
Customer Support Trace
3875 Airways Boulevard
Modul H, 4th Floor
Memphis, TN 38116

U.S. Mail: PO Box 727
Memphis, TN 38194-4643

Telephone: 901-369-3600

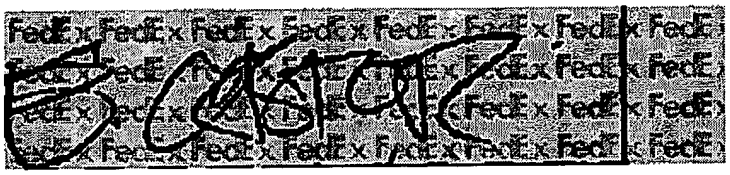
8/24/2000

Dear Customer:

Here is the proof of delivery for the shipment with tracking number **791836118067**. Our records reflect the following information.

Delivery Information:

Signed For By: E.CASTOR



Delivery Location: 2107 N 1ST ST 350
Delivery Date: August 21, 2000
Delivery Time: 1020

Shipping Information:

Tracking No: 791836118067

Ship Date: August 18, 2000

Recipient:
MR. DAVID KNAPP, CHIEF TEC. OF
GET2CHIP.COM, INC.
2107 NORTH FIRST STREET,
SAN JOSE, CA 95131
US

Shipper:
JONATHAN T. KAPLAN
BROWN RAYSMAN ET AL
120 W 45TH ST FL 21
NEW YORK, NY 100364041

Shipment Reference Information: 4000/10

Thank you for choosing FedEx Express. We look forward to working with you in the future.

FedEx Worldwide Customer Service
1-800-Go-FedEx®
Reference No.: R2000082400019752191

THIS PAGE BLANK (USPTO)

- Ship Inside U.S.
- Ship Outside U.S.
- Track Shipment
- Cancel Shipment
- Schedule Courier
- Address Book
- Shipping History

- Update User Profile
- Help/FAQs
- Tutorial
- Contact Information

- Go to fedex.com

FedEx has processed your shipment shown below. If you have any questions about this shipment, you can email us or contact your customer service representative.

Airbill Number : 791836118067

- **Delivered To : Recept/Frnt desk**
- **Delivery Location : SAN JOSE CA**
- **Delivery Date : 08/21/2000**
- **Delivery Time : 10:20**
- **Signed For By : E.CASTOR**
- **Status Exception : Pkg not due for delv/no attempt**
- **Scan Activity :**
  - **Delivered SAN JOSE CA 08/21/2000 10:20**
  - **Placed on Van SAN JOSE CA 08/21/2000 08:04**
  - **Pkg. Status Exception SAN JOSE CA 08/19/2000 09:18**
  - **Arrived at FedEx Destination Location SAN JOSE CA 08/19/2000 09:18**
  - **Arrived at Sort Facility OAKLAND CA 08/19/2000 08:22**
  - **Left FedEx Sort Facility MEMPHIS TN 08/19/2000 04:57**
  - **Left FedEx Sort Facility MEMPHIS TN 08/19/2000 02:49**
  - **Left FedEx Sort Facility MEMPHIS TN 08/19/2000 02:41**
  - **Left FedEx Sort Facility NEWARK NJ 08/18/2000 22:44**
  - **Arrived at Sort Facility MEMPHIS TN 08/19/2000 01:23**
  - **Left FedEx Sort Facility NEWARK NJ 08/19/2000 00:49**
  - **Left FedEx Origin Location NEW YORK NY 08/18/2000 22:14**
  - **Left FedEx Origin Location NEW YORK NY 08/18/2000 19:32**

Return to Shipping History | Return to Shipping

THIS PAGE BLANK (USPTO)

BROWN RAYSMAN MILLSTEIN FELDER & STEINER LLP

120 WEST FORTY-FIFTH STREET • NEW YORK, NY 10036 • TELEPHONE: 212 944 1515 • FACSIMILE: 212 840 2429

JONATHAN T. KAPLAN
PARTNER
212 827 9478
E-MAIL: jkaplan@brownraysman.com
WEB SITE: www.brownraysman.com

August 18, 2000

VIA FEDERAL EXPRESS

Mr. David W. Knapp
Chief Technical Officer
Get2Chip.com, Inc.
2107 North First Street, Suite 350
San Jose, CA 95131

Re: Reissue of U.S. Patent No. 5,764,951 to Ly et al. for METHODS FOR AUTOMATICALLY PIPELINING LOOPS
Our Ref.: 4000/10

Dear Mr. Knapp:

This is a follow up to our letter of 31 July 2000 requesting your review and signature for the above-identified reissue patent application. As we stated in our earlier letter, this law firm represents your former employer Synopsys, Inc. (hereinafter "Synopsys"), which is currently seeking reissuance of the above-identified patent in the United States Patent and Trademark Office. Enclosed please find a Declaration of Inventor which we will be filing in the Reissue Patent Application. Also enclosed is a copy of the Reissue Application, which adds new claims 23-34 to those which already issued in U.S. Patent 5,764,951. Please review the Declaration and the new claims carefully. The Declaration and Reissue Application are identical to those sent to you on 31 July. We have enclosed additional copies for your convenience.

Assuming you understand and agree with the Declaration, Synopsys requests that you do the following. Please complete the Declaration by entering, in the boxes at the end of the Declaration, your country of citizenship and the address of your current home residence. Once you have completed the Declaration, please sign the Declaration in the indicated box at the end of the Declaration. Return the Declaration to us by means of the enclosed self-addressed stamped envelope.

IT IS VERY IMPORTANT THAT YOU RETURN THE DECLARATION TO US BY SEPTEMBER 1, 2000. IF YOU HAVE ANY QUESTIONS ABOUT THE

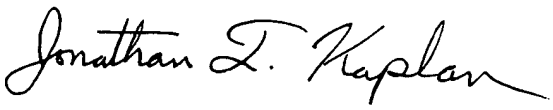
THIS PAGE BLANK (USPTO)

DECLARATION, IT IS EXTREMELY IMPORTANT THAT YOU CONTACT US AS SOON AS POSSIBLE SO THAT WE MAY ANSWER YOUR QUESTIONS.

As I am sure you will recall, you have previously assigned all rights in the invention to Synopsys as your former employer.

Your cooperation in this matter is greatly appreciated. On behalf of Synopsys, Inc., I thank you most sincerely for your help.

Very truly yours,



Jonathan T. Kaplan

JTK:MJM:od

Enclosures Declaration Of Inventor (w/ self-addressed stamped envelope)
Reissue Application

THIS PAGE BLANK (USPTO)

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Reissue Appl. No.	:	09/590,584
Filed	:	June 8, 2000
In re Patent to	:	Tai A. Ly et al.
Patent No.	:	5,764,951
Issued	:	June 9, 1998
Title	:	METHODS FOR AUTOMATICALLY PIPELINING LOOPS

Box REISSUE
Assistant Commissioner for Patents
Washington, D.C. 20231

DECLARATION OF INVENTOR DAVID W. KNAPP IN APPLICATION FOR BROADENING REISSUE OF PATENT
Pursuant to 37 CFR 1.63 and 1.175

This declaration is made in application for broadening reissue of the above-identified patent.

As a below named inventor, I hereby declare that:

My residence, post office address and citizenship are as stated below next to my name.

I believe I am an original, first and joint inventor of the subject matter which is claimed and for which a patent is sought on the invention entitled

METHODS FOR AUTOMATICALLY PIPELINING LOOPS

the specification of which: *(check one)*
☐ is attached hereto; or
☒ was filed on June 8, 2000 as U.S. Reissue Application Serial No. 09/590,584.

I hereby state that I have reviewed and understand the contents of the above identified specification, including the claims, as amended by any amendment referred to above.

I acknowledge the duty to disclose information which is material to patentability as defined in Title 37, Code of Federal Regulations, §1.56.

THIS PAGE BLANK (USPTO)

I believe the original patent to be wholly or partly inoperative or invalid, for the reasons described below:

☐ by reason of a defective specification or drawing.
☒ by reason of the patentee claiming more or less than he had the right to claim in the patent.
☐ by reason of other errors.

At least one error upon which this application for reissue is based is described as follows:

The limitation of claim 1 to methods comprising steps of parsing text descriptions including loops with delayed signal assignments having delay values and setting latencies of pipelines equal to said delay values is more limiting than necessary, and resulted in the patentee claiming less than he had a right to claim.

The limitation of claim 18 to systems comprising logic for parsing text descriptions including loops with delayed signal assignments having delay values and setting latencies of pipelines equal to said delay values is more limiting than necessary, and resulted in the patentee claiming less than he had a right to claim.

The limitation of claim 21 to computer program products comprising computer readable program code devices configured to cause a computer effect parsing of text descriptions including loops with delayed signal assignments having delay values and setting of latencies of pipelines equal to said delay values is more limiting than necessary, and resulted in the patentee claiming less than he had a right to claim.

All errors being corrected in this reissue application arose without any deceptive intention on the part of the applicant.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Full Name of First Joint Inventor	David W. Knapp		
Inventor's Signature		Date	
Residence		Citizenship	
Post Office Address			

THIS PAGE BLANK (USPTO)

METHODS FOR AUTOMATICALLY PIPELINING LOOPS

Related Applications

This application is related to U.S. patent application Ser. No. 08/440,101 entitled "Behavioral Synthesis Links to Logic synthesis" with inventors Ronald A. Miller, Donald B. MacMillen, Tai A. Ly and David W. Knapp filed on May 12, 1995, which is hereby incorporated by reference.

Background

Field of the Invention

This invention relates to the field of computer aided design for digital circuits, particularly to automatically pipelining loops in a behavioral synthesis system.

Statement of the Related Art

Behavioral Synthesis

Behavioral vs. Register Transfer Level Design

Many of today's integrated circuits are described using a Hardware Description Language (HDL). Two common HDL's are VHDL and Verilog. VHDL is described in the IEEE Standard VHDL Language Reference Manual available from the Institute of Electrical and Electronic Engineers in Piscataway, New Jersey which is hereby incorporated by reference. Verilog is described in The Verilog Hardware Description Language by Donald E. Thomas and Philip Moorby, Kluwer Academic Publishers, 1991 which is hereby incorporated by reference.

As integrated circuits become increasingly complex, hardware designers are increasingly using *synthesis* software to transform HDL descriptions of digital circuits into mapped logic. The designer writes a description of a digital circuit in VHDL, Verilog, or another HDL, and uses synthesis software to create a digital circuit from the description. Using synthesis software typically shortens the amount of time required to create a digital circuit from a design specification, and allows a designer to create more complex designs than is possible manually.

Many of today's complex designs are expressed as software descriptions and

simulated to verify their correctness. These designs are later translated from software into hardware, in the form of Integrated Circuits (ICs), Application Specific Integrated Circuits (ASICs), or Field Programmable Gate Arrays (FPGAs), for implementation in the final product. This design description methodology is called *algorithmic-level* design.

Instead of beginning design at the Register Transfer Level (RTL), behavioral synthesis begins at the algorithmic (behavioral) level. RTL level design is described in Computer Structures: Reading and Examples by C. Gorden Bell and Allen Newell, McGraw-Hill 1971. A behavioral hardware description language (HDL) specification contains instructions, operations, variables, and arrays similar to the original software algorithm.

The target architecture of behavioral synthesis is a general computing model that contains datapath, memory, and control elements. Conventional design techniques currently use a manual RTL design methodology to build a datapath. A datapath is a sequence of logic consisting of registers, higher order functional units (such as adders and multipliers), and multiplexers. The datapath in a digital circuit uses the circuit's inputs to compute output results. Registers are 1-bit memory elements which hold their value through each clock cycle.

Conventional design techniques also build a controller at the RTL to sequence and control the actions of the datapath, memory, and Input/Output (I/0). Frequently, such controllers are implemented using a Finite State Machine (FSM). Finite state machines are described in Switching and Finite Automata Theory by Zvi Kohavi, Computer Science Press, 1978 which is hereby incorporated by reference. Controllers may also determine actions such as which branch of a conditional statement is executed.

Behavioral synthesis builds this architecture by using automated methods of scheduling, allocation, register sharing, memory and control inferencing--all of which are performed manually in an RTL methodology. The designer is freed from having to specify the exact architecture of a design and can automatically explore many implementations to find the optimal architecture.

Components of Behavioral Synthesis

The High-Level Synthesis of Digital Systems by Michael McFarland, Alice Parker, and Raul Camposano, in Proceedings of the IEEE, February 1990, which is hereby incorporated by reference, provides an excellent overview of High Level Synthesis, as Behavioral Synthesis is often called.

Three components of a behavioral synthesis system are Scheduling, Allocation, and Resource Sharing.

Scheduling determines in which clock cycle each operation executes. Scheduling extracts the control and data flow operations of a design specification and assigns these operations to cycles. A state machine controller is synthesized to sequence the operations and execute them in their assigned cycle. The typical goal of this process is to assign operations to cycles so as to be able to implement the design with the fewest resources (registers, multiplexers, and operations) while at the same time minimizing the number of clock cycles (latency).

Allocation is a behavioral synthesis task that maps the operations and data of a behavioral HDL specification into the datapath, which contains memories, registers, functional units such as adders and multiplexers, and gates. Allocation determines which type of operation to use for each operator. For instance, if an operator performs addition, a ripple carry, a carry-lookahead, or some other type of adder can be used.

Resource Sharing attempts to share hardware resources between operators in a design. For example, consider two additions which occur in mutually exclusive conditional branches. Such additions will never be performed at the same time. Thus, they can be performed on the same piece of hardware. Resource sharing attempts to minimize the amount of hardware used by sharing hardware as much as possible.

Scheduling Modes

There are several modes for automatically scheduling operations into control steps. Briefly, these modes are cycle-fixed, superstate-fixed, and free-floating mode. In cycle-fixed mode, all I/O operations are constrained to occur in the same cycle in the original HDL descriptions and in the synthesized design. In cycle-fixed mode, the cycle

level behavior of the synthesized circuit must match the cycle level simulation behavior of the source HDL.

The other scheduling modes allow behavioral synthesis a greater degree of freedom in assigning states in a schedule. Scheduling modes are discussed further in Behavioral Synthesis Methodology for HDL-Based Specification and Validation by D. Knapp, T. Ly, D. MacMillen and R. Miller in Proceedings of the 31st DAC, June 1995, which is included as Appendix B and is hereby incorporated by reference. They are also discussed in Behavioral Compiler User Guide Version 3.2a available from Synopsys, Inc. in Mountain View, Calif., which is hereby incorporated by reference.

Loop Pipelining

In behavioral HDL, a loop repeatedly executes the operations in the loop body until an exit condition becomes true. Loop iterations are usually sequential; operations in the first iteration are executed, operations in the next iteration are executed, and so on, as shown in Figure 1. The throughput, that is the amount of data processed per unit time, of the function implemented by the loop body is limited by the critical path in the loop body.

In some loops, data required by an operation in the next loop iteration is available prior to completion of the current loop. Under these conditions, the designer can *pipeline* the loop--parallelizing execution of iterations to increase throughput beyond critical path limitations of the loop body. This process of loop pipelining schedules consecutive loop iterations to partially overlap in time; a new loop iteration is initiated before the current iteration has finished.

Figure 2 shows an example of loop pipelining where the data required by operation A in iteration two is available after operation C in the first loop iteration.

The two timing-related aspects of a loop that affect throughput are:

Initiation interval: The number of clock cycles between the start of two consecutive loop iterations.

Latency: The number of clock cycles required to execute all operations in a single loop iteration.

For sequential loops that are not pipelined, the initiation interval and latency of a loop are the same. For a pipelined loop, the initiation interval is smaller than the latency.

The primary reason for using loop pipelining is to increase the throughput of the design; the trade-off is that the design area usually increases.

Many designs have separate specifications on throughput and input-to-output delay. The throughput specification constrains the initiation interval. The input-to-output delay specification constrains the loop latency. Loop pipelining enables a flexible relationship between the initiation interval and latency of a loop.

An example of a candidate for loop pipelining is a design that processes a data stream. This type of design often has tight throughput requirements based on the rate of the data streams and loose input-to-output delay constraints.

Loop Carry Dependencies

Loop Carry Dependencies (LCDs) are data values produced in one iteration of a loop and consumed by operations in subsequent iterations.

In loop pipelining, loop iterations that are producers and consumers of LCDs can happen at the same time. To preserve data dependencies, the operations in a loop must be scheduled so that LCD values are available in time for the iteration in which they will be consumed. Two schedules for a LCD are shown in Figure 3.

The example of Figure 3(a) violates the LCD. Operation 410 is scheduled so that its output is not ready in time for operation 420 to use it in the next iteration of the loop. The example of Figure 3(b) is scheduled correctly. In this case, operation 410 is scheduled so that its output is ready in time for operation 420 to execute in the next iteration of the loop.

Memory and I/O Accesses

Loop Pipelining must preserve the original ordering of all reads and writes to the same memory, signal, or port. In addition, the ordering reads and writes in one iteration of the loop may not "cross," or occur after, reads and writes in subsequent iterations of the loop. Specifically, all reads and writes to the same memory, and all writes to the

same signal or port in one iteration of the loop must occur before any reads or writes to the same memory, signal or port in a subsequent iteration of the loop. All reads of the same signal or port must occur simultaneously to or before any read of the same signal or port in a subsequent iteration of the loop.

For example, Figure 4 shows two schedules for a loop that has two reads of signal x. In FIGURE 4(a), read 510 and read 520 are improperly scheduled. Read 520 occurs after read 510 occurs in the next iteration of the loop. In FIGURE 4(b), read 510 and read 520 are properly scheduled. In this schedule, read 520 occurs after read 510 in the next iteration of the loop.

A Brief Description of the Drawings

The accompanying drawings, which are incorporated in and constitute a part of this specification, illustrate several embodiments of the invention and, together with the description, serve to explain the principles of the invention.

Figure 1 shows an example of sequential loop processing.

Figure 2 shows an example of pipelined loop processing including the loop latency and initiation interval.

Figure 3 shows an example of a loop carry dependency.

Figure 4 shows an example of memory and I/O access restrictions in pipelined loops.

Figure 5 is a block diagram showing a computer system.

Figure 6 is a flowchart which shows steps in a circuit synthesis process.

Figure 7 is a flowchart which shows steps for scheduling preprocessing.

Figure 8 is a flowchart which shows steps for inserting constraints into a constraint graph.

Figure 9 is a flowchart which shows steps for scheduling templates.

Figure 10 is a flowchart which shows steps for creating a constraint using templates.

Figure 11 shows HDL source code which contains a loop with a producer and a consumer.

Figure 12 shows a circuit before scheduling which is created from loop 3030 of Figure 11.

Figure 13 shows a constraint created for a producer and consumer in loop 3030.

Figure 14 shows a circuit which is created after scheduling loop 3030 using an initiation interval of 2 and a latency of 4.

Figure 15 shows Verilog HDL source code which contains a loop with I/O dependencies.

Figure 16 shows a circuit before scheduling which is created from loop 1530 of Figure 15.

Figure 17 shows a constraint created for two reads in loop 1530.

Figure 18 shows a circuit which is created after scheduling loop 1530 using an initiation interval of 2 and a latency of 4.

Figure 19 (a) and Figure 19 (b) are examples of HDL source code including a delay clause.

Figure 20 is a flowchart showing steps performed during translation from the source code of Figure 19 (a) and Figure 19 (b) to a circuit design that incorporates a delay specified by the delay clause.

Figure 21 is a representation of a data flow graph generated from the source code of Figure 19 (a) and (b) in accordance with the steps of Figure 20.

Figure 22 is a representation of a control flow graph generated from the source code of Figure 19 (a) and (b) and the data flow graph of Figure 21.

Figure 23 is a flow chart showing steps performed to generate a control data flow graph from the control flow graph and data flow graph of Figure 21 and Figure 22.

Figure 24 is a representation of a control data flow graph generated by the steps of Figure 23.

Figure 25 is a diagram showing an example of loop tiling with and without the delay in the HDL.

Figure 26 is a diagram showing the effect of the delay clause on pipelining.

Figure 27 shows the operations of Figure 12 scheduled into control steps.

Figure 28 shows the read operations of Figure 16 scheduled into control steps.

Detailed Description of the Invention

The present invention is a method and apparatus for synthesizing a circuit which implements a pipelined loop from a Hardware Description Language (HDL) description. The following description is presented to enable any person skilled in the art to make and use the invention, and is provided in the context of a particular application and its requirements. Various modifications to the preferred embodiment will be readily apparent to those skilled in the art, and the generic principles defined herein may be applied to other embodiments and applications without departing from the spirit and scope of the invention. Thus, the present invention is not intended to be limited to the embodiment shown, but is to be accorded the widest scope consistent with the principles and features disclosed herein.

1.0 Computer System Description

Figure 5 illustrates a computer system 100 in accordance with a preferred embodiment of the present invention. The computer system 100 includes a bus 101, or other communications hardware and software, for communicating information, and a processor 109, coupled with the bus 101, is for processing information. The processor 109 can be a single processor or a number of individual processors that can work together. The computer system 100 further includes a memory 104. The memory 104 can be random access memory (RAM), or some other dynamic storage device. The memory 104 is coupled to the bus 101 and is for storing information and instructions to be executed by the processor 109. The memory 104 also may be used for storing temporary variables or other intermediate information during the execution of instructions by the processor 109. The computer system 100 also includes a ROM 106 (read only memory), and/or some other static storage device, coupled to the bus 101. The ROM 106 is for storing static information such as instructions or data.

The computer system 100 can optionally include a data storage device 107, such as a magnetic disk, a digital tape system, or an optical disk and a corresponding disk drive. The data storage device 107 can be coupled to the bus 101.

The computer system 100 can also include a display device 121 for displaying

information to a user. The display device 121 can be coupled to the bus 101. The display device 121 can include a frame buffer, specialized graphics rendering devices, a cathode ray tube (CRT), and/or a flat panel display. The bus 101 can include a separate bus for use by the display device 121 alone.

An input device 122, including alphanumeric and other keys, is typically coupled to the bus 101 for communicating information, such as command selections, to the processor 109 from a user. Another type of user input device is a cursor control 123, such as a mouse, a trackball, a pen, a touch screen, a touch pad, a digital tablet, or cursor direction keys, for communicating direction information to the processor 109, and for controlling the cursor's movement on the display device 121. The cursor control 123 typically has two degrees of freedom, a first axis (e.g., x) and a second axis (e.g., y), which allows the cursor control 123 to specify positions in a plane. However, the computer system 100 is not limited to input devices with only two degrees of freedom.

Another device which may be optionally coupled to the bus 101 is a hard copy device 124 which may be used for printing instructions, data, or other information, on a medium such as paper, film, slides, or other types of media.

A sound recording and/or playback device 125 can optionally be coupled to the bus 101. For example, the sound recording and/or playback device 125 can include an audio digitizer coupled to a microphone for recording sounds. Further, the sound recording and/or playback device 125 may include speakers which are coupled to digital to analog (D/A) converter and an amplifier for playing back sounds.

A video input/output device 126 can optionally be coupled to the bus 101. The video input/output device 126 can be used to digitize video images from, for example, a television signal, a video cassette recorder, and/or a video camera. The video input/output device 126 can include a scanner for scanning printed images. The video input/out-put device 126 can generate a video signal for, for example, display by a television.

Also, the computer system 100 can be part of a computer network (for example, a LAN) using an optional network connector 127, being coupled to the bus 101. In one

embodiment of the invention, an entire network can then also be considered to be part of the computer system 100.

An optional device 128 can optionally be coupled to the bus 101. The optional device 128 can include, for example, a PCMCIA card and a PCMCIA adapter. The optional device 128 can further include an optional device such as modem or a wireless network connection.

2.0 Definitions

A digital circuit is an interconnected collection of parts. Parts may also be called cells. The digital circuit receives signals from external sources at points called primary inputs. The digital circuit produces signals for external destinations at points called primary outputs. Primary inputs and primary outputs are also called ports. Each part receives input signals and computes output signals. Each part has one or more pins for receiving input signals and producing output signals. In general, pins have a direction. Most pins are either input pins, which are called loads, or output pins, which are called drivers. Some pins may be bidirectional pins, which can be both drivers and loads.

Two or more pins from one or more parts or primary inputs or primary outputs are connected together with a net. Each net establishes an electrical connection among the connected pins, and allows the parts to interact electrically with each other. Pins are also connected to primary inputs and primary outputs with nets. For the sake of simplicity, parts may be said to be "connected" to nets, but it is actually pins on the parts which are connected to the nets.

A Circuit Element is any component of a circuit. Ports, pins, nets, and cells are all circuit elements. Any circuit element which is an input to another circuit element is said to drive that circuit element. Any circuit element which is an output of another circuit element is said to load that circuit element. For example, drivers drive a signal onto a net; loads load nets with capacitance.

A digital circuit design can be stored in memory of a computer system using data structures which represent the various components of the circuit. The data structures have the same name as the physical components. In this document, parts,

cells, nets, pins, and other digital circuit components refer to the software representation of the physical digital circuit component.

A digital circuit can be specified hierarchically. Some or all of the parts in the digital circuit may themselves be digital circuits composed of more interconnected parts. When a high level part is specified as a digital circuit composed of other, lower level parts, the pins of the high level part become the primary inputs and primary outputs for the digital circuit comprising the lower level parts. When a high level part is composed of lower level parts, it is called a level of hierarchy.

Following are additional definitions of terms which are used in this document.

An *HDL* is a Hardware Description Language. HDL's are used to describe designs for digital circuits.

A *Translated Circuit, Generic Technology Circuit, or GTech Circuit* is a software representation of a digital circuit which does not include references to a specific technology, but rather refers to cells that implement generic logic such as "and", "or", and "not". This software representation is stored in memory 104 of computer system 100.

A *Mapped Circuit* is a software representation of a digital circuit which is built from parts available in a technology library which is provided by a silicon vendor. This software representation is stored in memory 104 of computer system 100. A mapped circuit can be timed using a conventional timing verifier such as DesignTime, available from Synopsys, Inc. in Mountain View, Calif. After it is built, a netlist representation of a mapped circuit can be sent to a silicon vendor for layout and fabrication. For instance, the mapped circuit can be written out using LSI netlist format and sent to LSI Logic in Milpitas, Calif. The process of creating a mapped circuit from a generic technology circuit is called mapping. Because a circuit must be mapped before it can be timed, mapped circuits are also used internally by synthesis tools.

The *Fanout* of a circuit element includes any circuit elements which are driven by that circuit element. The *transitive fanout* of a circuit element includes all of the circuit elements in the circuit which are driven, either directly or indirectly, by that

circuit element. Thus, the transitive fanout of a circuit element includes the fanout of that circuit element, as well as the fanout of each of the circuit elements in the original fanin, and so on.

The *Fanin* of a circuit element includes any circuit elements which drive that circuit element. The *transitive fanin* of a circuit element includes all of the circuit elements in the circuit which drive, either directly or indirectly, that circuit element. Thus, the transitive fanin of a circuit element includes the fanin of that circuit element, as well as the fanin of each of the circuit elements in the original fanin, and so on.

An *Operator* is a function, such as addition. Such functions are used in HDL source code. For example, the plus in "c=a+b;" is an operator.

An *Operation* is a software representation of a hardware functional unit which performs a function such as addition. For example, a software representation of an adder is an operation.

A *Clock Cycle* is a period of time, for example 10ns, between pulses of a clocking element in a digital circuit. The clocking element is used to synchronize the digital circuit.

3.0 Scheduling

Scheduling is a well defined problem which has been studied extensively. An overview of the scheduling problem is available in The High-Level Synthesis of Digital Systems by Michael McFarland, Alice Parker, and Raul Camposano, in Proceedings of the IEEE, February 1990, which is hereby incorporated by reference.

The input to a scheduler is typically a set of hardware operations, a set of constraints between the hardware operations, a clock period, and a set of control steps into which the hardware operations must be mapped. The output is a schedule where each hardware operation is mapped to a control step.

Schedulers typically use a number of graphs. For instance, the constraints for a scheduler are often represented using a graph. Nodes in the graph typically represent events to be scheduled, such as operations, and edges in the graph represent constraints between the events. The scheduler checks the constraint graph to ensure that all of the

constraints are met before placing an event into a particular control step. Schedulers also use control graphs, data flow graphs, and combination control data flow graphs (CDFG's). Control graphs represent the flow of control in a circuit. Data flow graphs represent the flow of data in a circuit; that it the flow of data from the inputs to the outputs of the circuit. Control data flow graphs combine both control flow and data flow information into a single graph. All of these types of graphs are described in High-Level Synthesis (subtitled Introduction to Chip and System Design) by Daniel Gajski, Nikil Dutt, Allen C-H Wu, and Steve Y-L Lin, Kluwer Academic Publishers, 1992 which is hereby incorporated by reference and will subsequently be referred to as High-Level Synthesis by Gajski et al.

An additional technique used for scheduling circuits involves "templates". Templates are described in Scheduling using Behavioral Templates by Tai Ly, David Knapp, Ron Miller, and Don MacMillen in Proceedings of the 31st DAC, June 1995, which is included as Appendix A and is hereby incorporated by reference. Simply speaking, templates are data structures which specify scheduling constraints among CDFG nodes. Templates "lock" the control step relationship between 2 or more CDFG nodes. Figure 13 shows an example of two templates, template 1250 and template 1280. Each template contains one or more nodes, some of which may represent operations. For example, adder node 2020 represents adder 3120 of Figure 12.

3.1 Overview of Synthesis with Scheduling

Figure 6 is a flowchart showing how scheduling steps fit into the overall synthesis strategy. This flowchart shows how a mapped circuit is created from a source HDL description. The input to synthesis is an HDL description of a digital circuit. Such a description may be written in VHDL, Verilog, or some other HDL.

An HDL description is translated in step 810 to generic logic. A conventional HDL translator 1310 such as VHDL Compiler version 3.2b from Synopsys, Inc. in Mountain View, Calif. preferably is used.

Step 820 performs scheduling preprocessing steps. These steps are shown in Figure 7 and Figure 8.

Step 830 schedules the operations in the circuit. A method for scheduling the operations in the circuit is shown in Figure 9.

Step 840 netlists the scheduled circuit. Netlisting creates a GTech circuit from the scheduled CDFG. The CDFG representation of the circuit in memory is transformed into a GTech representation of the circuit in memory.

In step 850, the resulting GTech circuit is optimized using conventional logic synthesis such as Design Compiler version 3.2 b by Synopsys, Inc. in Mountain View, Calif. The output of logic optimization is a mapped circuit description which can be sent to a silicon vendor for fabrication. For example, a description of the mapped circuit can be output using LSI Netlist format and sent to LSI Logic in Milpitas, Calif for fabrication.

3.2 Scheduling Preprocessing

Figure 7 is a flowchart which shows steps for scheduling preprocessing. The input to the method is an annotated GTech circuit. Annotations on the circuit include delayed signal assignment information. The use of delayed signal assignments will be discussed in a later section.

Step 910 extracts a control graph from the annotated GTech using conventional techniques. In addition, information concerning delayed signal assignments is extracted as described below.

Step 920 extracts a Control Data Flow Graph (CDFG) from the control graph created in step 910 and the data flow graph represented by the GTech circuit. This is also done using conventional techniques.

Step 930 creates initial templates for the operations in the CDFG as described in Scheduling Using Behavioral Templates in Appendix A. These initial templates form the initial constraint graph.

Step 940 inserts constraints in the constraint graph. Some types of constraints are discussed in Scheduling Using Behavioral Templates in Appendix A. Other types of constraints are a part of the present invention and will be discussed in subsequent sections.

3.3 Inserting Constraints

Step 940 of Figure 7 is implemented by Figure 8 which is a flowchart which shows steps for inserting constraints into a constraint graph which uses templates. The input to the process is a CDFG and a constraint graph.

Step 1110 identifies Loop Carry Dependency (LCD) producer consumer pairs. LCD's are identified by tracing the CDFG using conventional techniques. LCD's are discussed below in connection with Figure 11, Figure 12, Figure 13, Figure 14, and Figure 27.

Step 1120 constrains the LCD's. Constraining LCD's involves adding constraints to the constraint graph so that producer and consumer operations are scheduled so that the consumer consumes a value produced by the producer before it is overwritten in a subsequent iteration of the loop. A method and apparatus for constraining LCD's will be discussed in a later section.

Step 1130 identifies memory and I/O access dependencies in loops which will be scheduled using pipelines. I/O accesses include reads and writes to memories, signals, and ports. Reads and writes in one iteration of the loop may not "cross," or occur after, reads and writes in subsequent iterations of the loop. Specifically, all reads and writes to the same memory, and all reads and writes to the same signal or port in one iteration of the loop must occur before any reads or writes to the same memory, signal or port in a subsequent iteration of the loop. The one exception to this rule is that reads of the same signal or port may occur simultaneously to a read of the same signal or port in a subsequent iteration of the loop. This step finds the first and last accesses for each memory, signal, or port by tracing through the CDFG using conventional techniques. Memory and I/O accesses are discussed below in connection with Figure 15, Figure 16, Figure 17, Figure 18, and Figure 28.

Step 1120 constrains the memory and I/O accesses in pipelined loops. Constraining memory and I/O accesses involves adding constraints to the constraint graph so that first and last accesses are scheduled so that the last access occurs before the first access in a subsequent iteration of the loop. A method and apparatus for

constraining memory and I/O accesses will be discussed in a later section.

Step 1130 inserts other types of constraints into the constraint graph. Such constraints are discussed in Scheduling Using Behavioral Templates in Appendix A. An example of another type of constraint is a dataflow constraint, which ensures that data values are produced before they are consumed by subsequent operations.

3.4 Scheduling Templates

Figure 9 is a flowchart which shows steps of scheduling (step 830 of Figure 6) using templates. The input to the process is the CDFG and the constraint graph created by the steps of Figure 8. It is possible to schedule templates using many different scheduling techniques. A number of scheduling techniques are described in High-Level Synthesis by Gajski et al, particularly in Chapter 7. This figure shows a general method, which is provided as an example.

Step 1010 creates the As Soon As Possible (ASAP) and As Late As Possible (ALAP) schedules for each template while satisfying the constraints represented in the constraint graph. The ASAP schedule places each template into the earliest possible control step (c-step). The ALAP schedule places each template into the latest possible control step. Together, the earliest and latest control steps define a range into which each template may be scheduled. A method for determining the ASAP and ALAP schedules for templates is described in Scheduling Using Behavioral Templates in Appendix A.

Loop 1020 loops until a "good" schedule is found. A "good" schedule is one which fulfills the constraints specified in the constraint graph and optimizes for a specific goal specified by a human designer, such as fewest number of control steps. Different scheduling techniques use different criteria for deciding when to stop trying to improve the schedule. For example, one technique might stop when the constraints are all met, or when a certain amount of CPU time has been spent, whichever comes last.

Step 1030 picks a template in the constraint graph to schedule. Different techniques use different criteria for deciding what to schedule next. Generally, template scheduling techniques use criteria based upon the operations in a template. For instance,

a list scheduling technique which uses priorities will assign a priority to a template based on the priorities of the operations within the template. (List scheduling is described in High-Level Synthesis by Gajski et al in Chapter 7).

Step 1040 schedules the chosen template in the control step chosen by the scheduling technique being used. Templates are scheduled by placing the first operation within the template into the chosen control step and the remaining operations within the template into subsequent control steps as defined by the template.

Arrow 1050 indicates that loop 1020 iterates until a "good" schedule is found.

4.0 Method for Creating Constraints

This section describes a general technique for constraining the relationship between two nodes in a constraint graph. Such constraints are added in step 940 of Figure 7. The section then describes examples of using this technique to constrain loop carry dependencies and I/O dependencies.

4.1 Placeholder Node Method

Figure 10 shows a general method for creating a scheduling constraint between two nodes in a constraint graph. Such constraints are created in step 1120 and step 1140 of Figure 8 to constraint LCD's and memory and I/O accesses. This section shows a general method and discusses specific examples. The first example constrains an LCD; the second example constrains a pair of signal reads. The input to the process of Figure 10 is a constraint graph, two templates in the graph, Event 1 and Event 2, an integer n, and a number of cycles c. "n" is the number of cycles within which Event 2 must be scheduled after Event 1. "c" is either 0 or 1. "c" has value 0 when Event 2 must be schedule before n cycles after Event 1, and value 0 when Event 2 may be scheduled exactly n cycles after Event 1.

Step 610 adds a placeholder node H to the template for Event 1 in the constraint graph. A placeholder rode is a node in the constraint graph which is only used to create constraints. The placeholder node does not represent any portion of the final circuit. Placeholder node H is inserted into the Event 1's template such that it is locked n cycles after Event 1.

Step 620 adds a constraint in the constraint graph from Event 2 to placeholder node H which constrains Event 2 to occur c cycles before placeholder node H, where c is 0 or 1. The value of c depends on the constraint being added and will be discussed in subsequent sections.

4.2 Using Placeholder Nodes for Loop Carry Dependencies

The following section provides an example of constraining loop carry dependencies using placeholder nodes. Such constraints are created in step 1120 of Figure 8. A loop carry dependency is a data value which is produced in one iteration of a loop and consumed by operations in subsequent iterations of the loop. To use the placeholder node method to schedule loop carry dependencies, Event 1 is set to be the operation which consumes the data. Event 2 is set to be the operation which produces that data. Event 2 must be scheduled so that the correct data values are driving it when it feeds its outputs to Event 1. If the consumer (Event 1) consumes the data one iteration after the producer (Event 2) creates it, then n is set to be the initiation interval of the loop. If the consumer consumes the data k iterations after it is created by the producer, then n is set to be k * initiation interval. For LCD's, "c" has value "1" because the producer must be scheduled before the consumer in the subsequent iteration of the loop.

Figure 11 shows an example of Verilog source code for a loop 3030 with a loop carry dependency between addition 3020 and subtraction 3010. The output of addition 3020, p, drives the input of subtraction 3010 on the next iteration of the loop. "p" is a Loop Carry Dependency. In this example, a human designer has specified that loop 3030 will be scheduled using an initiation interval of 2 and a latency of 4. Although this loop would not usually be pipelined because pipelining does not increase its throughput, this simple example is used for the sake of clarity.

Figure 12 shows a GTech circuit representation 2000 which is created for loop 3030 in Figure 11. The GTech circuit representation is stored in memory 104. GTech circuit 2000 is output from step 810 of Figure 6. Addition 3020 is implemented as adder 3120, and subtraction 3010 is implemented as subtracter 3110. Port p 2040 drives subtracter 3110. Port p' 2045 is driven by adder 3120. Port p 2040 and port p' 2045 are

partner ports. Partner ports are ports which represent the same signal, and thus frequently embody loop carry dependencies. Partner ports contain references to their partners. In the described embodiment, these references are implemented as pointers. Each port which has a partner contains a pointer to its partner port.

Figure 13 shows a constraint 1270 between adder node 2020, which is the producer for this LCD, and subtracter node 2010 which is the consumer of this LCD. The consumer and producer were identified in step 1110 of Figure 8. This constraint is created using the method of Figure 10. The starting templates are shown in Figure 13(a). First step 610 of Figure 10 adds placeholder node H 2060 to the template 1250 of subtracter node 2010. Because the initiation interval for the loop is 2, placeholder node H 2060 is constrained to be 2 cycles after subtracter node 2010 by template 1250. Next, step 620 creates constraint 1270, represented by an arrow, which constrains adder node 2020 to be at least one cycle before placeholder node H 2060. The modified templates and the new constraint are shown in Figure 13(b). The new constraint is then used to schedule the loop correctly using a method such as the one shown in Figure 9.

Figure 27 shows the add and subtract operations of Figure 12 scheduled into control steps by step 830 of Figure 6. For the sake of clarity, the other operations in the circuit are not shown. Two iterations of the loop are shown, to demonstrate how the schedule properly handles the loop carry dependency. Adder 3120 is scheduled so that its result is available before subtracter 3110 needs it in the next iteration of the loop.

Figure 14 shows the circuit created from the Verilog HDL source code of Figure 11 after scheduling. Block 3190 represents the representation of the FSM controller for this circuit stored in memory 104.

4.3 Using Placeholder-Nodes for I/O Dependencies

Loop pipelining must preserve the original order of all reads and writes to the same memory, signal, or port. The placeholder node method can be used to create constraints which ensure that I/O accesses in different iterations of the loop do not cross one another. Such constraints are created in step 1140 of Figure 8. The last I/O access to the same memory, signal, or port in a loop must occur simultaneously to or before the

first I/O access to that memory, signal or port in the next iteration of the loop. Specifically, reads of the same signal or port may occur simultaneously with reads in the next iteration of the loop, but not after. Writes to the same signal or port must occur before any read or write to the same signal or port in the next iteration of the loop. Reads and writes to the same memory must occur before any read or write to the same memory in the next iteration of the loop.

Thus, any last I/O access must occur within the initiation interval of the first I/O or memory access. To create this constraint, Event 1 of Figure 10 is set to be the first I/O access to a given memory, signal or port. Event 2 of Figure 10 is set to be the last I/O access to a given memory, signal or port. n is set to be the initiation interval of the loop, and c is set to be 0 or 1. Specifically, c is set to be 0 if Event 1 and Event 2 are signal or port reads. c is set to be 1 if Event 1 or Event 2 are signal or port writes, or memory reads or writes.

Figure 15 shows an example of Verilog source code for a loop 1530 with an I/O dependency between read 1510 and read 1520. Both read 1510 and read 1520 read the value of the same signal, x. Thus, read 1520 must be scheduled such that it occurs before read 1510 in the next iteration of the loop. In this example, a human designer has specified that this loop 1530 will be scheduled using an initiation interval of 1 and a latency of 3.

Figure 16 shows the GTech circuit 1500 which is created for loop 1530 of Figure 15. Circuit 1500 is output from step 810 of Figure 6. Read 1510 is implemented by read operation 3130. Read 1520 is implemented by read operation 3140. In this example, a human designer has specified that this loop will be pipelined with an initiation interval of 1 and a latency of 3.

Figure 17 shows a constraint between read node 1610, the first read of x in loop 1530, and read node 1620, the last read of x in loop 1530. Read node 1610 and read node 1620 were identified in step) 1130 of Figure 8. This constraint is created using the method of Figure 10. First step 610 adds placeholder node H 1760 to the template 1750 of read node 1610. Placeholder node H is constrained to be 1 cycle after read node

1610, because the initiation interval is 1, by template 1650. Next, step 620 creates constraint 1770, represented by an arrow, which constrains read node 1620 to be at least 0 cycles before, that is in the same cycle or after, placeholder node H 1760. Read node 1620 is constrained to be 0 cycles before placeholder node H 1760 because read node 1620 and read node 1610 are both signal reads, and as such are allowed to occur in the same control step. Constraint 1770 is then used to schedule the loop correctly using a method such as the one shown in Figure 9.

Figure 28 shows read operations on signal x of Figure 16 scheduled into control steps by step 830 of Figure 6. For the sake of clarity, the other operations in the circuit are not shown. Two iterations of the loop are shown, to demonstrate how the schedule properly handles the multiple signal reads. Read 3130 is scheduled so that it occurs simultaneously with read 3140 in the next iteration of the loop. Since simultaneous signal reads are allowed, this is a legal schedule.

Figure 18 shows the circuit created from the Verilog HDL source code of Figure 11 after scheduling.

5.0 Circuit Synthesis using Delayed Signal Assignment Information

Conventional design methodology uses a simulator to verify the correctness of a design both before and after it is synthesized. Conventional simulation systems, especially those systems performing behavioral synthesis, do not always yield identical cycle timing characteristics when HDL source code is simulated and when a synthesis output (a representation of a synthesized circuit) is simulated. It is advantageous for behavioral synthesis to be able to infer a circuit which will have the same cycle by cycle behavior during simulation as the simulation of the source HDL.

The source code of Figure 19(a) is written in the Verilog circuit specification language. The source code of Figure 19(b) is written in the VHDL circuit specification language. Both Verilog and VHDL are Hardware Description Languages (HDLs).

In Figure 19(a), the Verilog source code includes a signal assignment statement:

c<=#24x-p;

This statement includes a delay clause ("#24") indicating that a delay of twenty-

four time units, e.g., nanoseconds, should pass before the write operation is performed by the circuit that is to be generated. The delay clause is an example of delayed signal assignment information. Note that the inclusion of the delay clause in the HDL indicates a delay of the write operation only. The delay clause does not cause a delay in the performance of the subtraction operation. Similarly, in Figure 19(b), the VHDL source code includes a signal assignment statement:

c<=transportx-p after 24 ns;

This statement also contains a delay clause ("after 24 ns") indicating that a delay of twenty-four time units should occur in the generated circuit before the write operation is performed. This delay clause is a further example of delayed signal assignment information.

A circuit loop generated from the HDL source code of Figure 19(a) and Figure 19(b) will have an initiation interval of "2" because each source code example has two "wait" (or "posedge" or "negedge") statements within the loop. As discussed below, the delay clause in the source code causes the resulting loop to have a loop latency of "4". Figure 19(a) and Figure 19(b) are included for the purpose of example only. The present invention can use any appropriate type of source code (VHDL, Verilog, etc.) to represent a delay clause.

Figure 20 is a flowchart showing steps performed during translation step 810 of Figure 6 to generate a cdb. The exact placement of the steps of Figure 20 are not a part of the present invention and the steps also can be performed, for example, in the preprocessing step 820 of Figure 6. The input to Figure 20 is a representation of one of the source code examples of Figure 19(a) and Figure 19(b), such as a parse tree generated from the source code. The steps of Figure 20 are performed for each statement in the source code. The output of the translation step 810 and Figure 20 is a data flow graph (a "Gtech circuit") and a control flow graph (a "control data base" (cdb)). It will be understood by persons of ordinary skill in the art that the steps of Figure 20 and Figure 23 are performed by processor 109 of Figure 5, performing instructions stored in memory 104 of Figure 5.

In step 2002, the processor determines whether the current source code statement is a signal assignment statement (e.g., an assignment to a port using the "<=" operator) that includes a delay clause (e.g., "#24" in Verilog or "after 24 ns" in VHDL). If not, in step 2002, the processor performs standard processing for the node to build a node in the data flow graph. If the current source code statement includes a delay clause, then, in step 2004, the processor builds a write operation node in the data flow graph and annotates the node by adding an attribute indicating delayed signal assignment information to show that the write operation corresponding to the write operation node has a delay of, e.g., 24 nanoseconds (see node 2114 of Figure 21 and Figure 22).

Figure 21 shows an example of a data flow graph 2100 generated from one of the source code examples of Figure 19(a) and Figure 19(b) in accordance with the steps of Figure 20. A representation of data flow graph 2100 is stored in memory 104. Data flow graph 2100 includes as inputs a port x, a register p, and ports y and z. Each port has zero or more read operation nodes ("read op") 2102, 2104, 2106 associated therewith and each read operation node has an attribute indicating a port name (e.g., "port=`x`"). Respective ones of the inputs are input to a subtracter node 2110 and an adder node 2112. Subtracter node 2110 is connected to a write operation node 2114. Adder node 2112 is connected to a variable assignment node 2116. Output p' is input as p during successive iteration of the loop. Thus, the data flow graph of FIGURE 21 has seven nodes representing the data flow in the circuit to be synthesized.

In step 2008 of Figure 20, if there are more statements in the source code, control returns to step 2002. If all statements have been processed and a data flow graph (including signal delay attributes) has been generated for the source code, control passes to step 2012, where a control flow graph, such as that in Figure 22 is created.

Control graph 2200 of Figure 22 adds control information to nodes 2102, 2104, 2106, 2110, 2112, 2114, and 2116 indicating the order and conditions under which the data flow nodes are executed in the synthesized circuit. A representation of control graph 2200 is stored in memory 104 of Figure 5. The present invention preferably

operates in a "cycle fixed mode" in which each "wait" (or "posedge" or "negedge") statement in the source code indicates a new cycle in the synthesized circuit. Various processes for generating of control flow graphs are known to person of ordinary skill in the art and are described in High-Level Synthesis by Gajski et al.

In Figure 22, cnodes are used as "placeholder" nodes in the control graph to represent a collection of data flow nodes. Thus, cnode 2200 is associated with write operation node 2114 (including the signal delay attribute), read operation node 2102, and subtracter node 2110. The wait nodes in Figure 22 are used to represent the transitions between each cycle (or "cstep"). A wait node 2204 is used to mark the transition between the first cstep (cstep 0) and the second cstep (cstep 1). Wait node 2204 also has attributes indicating that it is based on a rising clock edge (due to the "posedge" statement in the source code) "Wait statements" (in VHDL source code) are treated similarly. Cnode 2206 (located in the second cstep) is associated with variable assignment node 2116, read operation node 2104, read operation node 2106, and adder node 2112. The control graph also includes a second wait node 2208 and a third cnode 2210.

As shown in Figure 7, the control flow graph is input to step 920, where a control data flow graph (CDFG) is created. The general procedure for creating a conventional CDFG is known to person of ordinary skill in the art and is described in High-Level Synthesis by Gajski et al. Figure 23 shows certain details of the process of creating a CDFG that relate to the delay clause of the present invention. An example CDFG is shown in Figure 24. The steps of Figure 23 are performed for each loop in the control flow graph. In step 2302, the processor sets a Wait.sub.-- count variable and a Max.sub.-- wait.sub.-- count variable in the memory 104 to an initial value of "0". In step 2304 the processor builds a "loop begin" node in the CDFG and assigns to it a cstep attribute value equal to "0".

Step 2306 is a first step in a loop performed by the processor for each cdb node. In step 2308, if the current cdb node is a cnode, control passes to step 2310, which is a first step in a loop performed for all data flow nodes associated with the current cdb

node. In step 2312, if a current data flow node is a write operation node having a delay clause (i.e., if the current data flow node represents a delayed signal assignment), control passes to step 2322.

In step 2322, a temp.sub.-- wait.sub.-- count variable is set to the current value of Wait.sub.-- count + a number of delay time units in the delayed signal assignment divided by the clock period (e.g., 0+24/6=4). A CDFG node is created and assigned to cstep temp.sub.-- wait.sub.-- count in step 2324. In step 2326, if temp.sub.-- wait.sub.-- count is greater than Max.sub.-- wait.sub.-- count, then in step 2328, Max.sub.-- wait.sub.-- count is set equal to temp.sub.-- wait.sub.-- count. Otherwise, control passes to step 2342. If, in step 2342, there are more data flow nodes associated with the current cdb node, then control passes to step 2310. Otherwise control passes to step 2336.

If, in step 2312, the current data flow nodes not a delayed signal assignment, the processor builds a standard CDFG node in step 2314 and assigns the created data flow node to cstep wait.sub.-- count in step 2316. If, in step 2318, wait.sub.-- count is greater than Max.sub.-- wait.sub.-- count, then Max.sub.-- wait.sub.-- count is assigned to wait.sub.-- count in step 2320. Control next passes to step 2342.

If, in step 2308, the current cdb node is not a cnode, then control passes to step 2330. If in step 2330 the current cdb node is a wait node, then wait.sub.-- count is incremented in step 2332 and control passes to step 2336. If, in step 2330, the current cdb node is not a wait node, then regular processing is performed to create a CDFG node in step 2334 and control passes to step 2336.

In step 2336, if there are more cdb nodes to process, then control passes to step 2306. Otherwise, a loop.sub.-- latency variable in memory 104 for the loop is assigned to Max.sub.-- wait.sub.-- count and an initiation interval variable for the loop is assigned to wait.sub.-- count in step 2338. In step 2340, the processor builds a "loop end" node in the CDFG and assigns it to cstep wait.sub.-- count.

The output of step 920 of Figure 7 is input to the scheduler, which uses the CDFG and the loop initiation interval and loop latency to schedule the nodes of the circuit being generated. In the described embodiment, all nodes except read/write

operation nodes can "float", i.e., can be moved between csteps by the scheduler to allow the scheduler to create an efficient circuit design. In the CDFG, these nodes are always assigned a cstep value equal to the initial cteps in which they appear in the HDL as a "suggestion" to the scheduler. It will be understood by persons of ordinary skill in the art that the CDFG of Figure 24 has been simplified for the sake of example and that the CDFG also includes, e.g., data flow arcs connecting the CDFG nodes that represent data flows in a similar manner to the data flows of Figure 21.

Figure 14 shows an example circuit synthesized from the CDFG of Figure 24. Figure 25 shows an example of placement of CDFG nodes in csteps without and with use of the delay clause. In the left column, which represents CDFG without the delay clause, CDFG nodes corresponding to write operation node 2114, read operation node 2109, and subtracter node 2110 are assigned to cstep 0. Similarly, CDFG nodes corresponding to adder node 2112, read operation node 2104, read operation node 2106, assignment node 2116 (and a CDFG loop.sub.-- end node) are assigned to a second cstep 1. Generation of this CDFG representation causes the synthesizer to generate a circuit that has different timing characteristics than the characteristics generated by the circuit synthesizer when the source code includes a delay clause. The right column of Figure 25 shows the assignment of CDFG nodes to cycles in accordance with the present invention. In this example, a write operation node corresponding to write operation node 2114 is moved into cstep 4 during the steps of FIGURE 23. This modification of the process to generate the CDFG (possible because of an addition of a signal delay attribute to the data graph 2100) allows the synthesis process to generate a circuit that has cycle level simulation behavior that is substantially identical to that of the cycle level simulation behavior of the source HDL.

Figure 26 shows an example of loop pipelining when the present invention is used. The figure shows an nth iteration of the loop and an n+1st iteration of the loop over time. As can be seen in the figure, the initial interval of successive iterations of the loop is equal to a number of wait statements (or "posedge" or "negedge" statements). The loop latency, is equal to the longest cycle delay from the beginning of the loop to a

latest operation. The throughput of the pipelined loop is not decreased by use of delayed signal assignments. In general, the scheduler will schedule a circuit having the CDFG of Figure 24 as a pipelined circuit because the loop latency is longer than the initiation interval.

In summary, use of delayed signal assignments allows behavioral synthesis to infer circuits with pipelined loops which have cycle level simulation behavior which matches that of the source HDL. Pipelined loops may include loop carry dependencies and/or I/O and/or memory accesses which must be scheduled correctly. The use of a placeholder node within a template is an efficient representation of such scheduling constraints.

WHAT IS CLAIMED IS:

1. A method performed by a data processing system having a memory, comprising the steps of:

parsing a text description of a circuit, said text description stored in the memory, said text description including a loop with a delayed signal assignment having a delay value;

translating said text description into a digital circuit representation in said memory, said digital circuit representation including a pipeline; and

setting a latency of said pipeline equal to said delay value.

2. The method of claim 1, wherein said loop further includes N wait statements, where N is greater than zero, said method further comprising the step of setting an initiation interval of said pipeline equal to N.

3. The method of claim 1, wherein said text description is written in Verilog and said delayed signal assignment uses a Verilog "#" operator.

4. The method of claim 3, wherein said wait statements use Verilog "@posedge" statements.

5. The method of claim 3, wherein said wait statements use Verilog "@negedge" statements.

6. The method of claim 1, wherein said text description is written in VHDL, said delayed signal assignment uses a VHDL "after" clause, and said wait statements use VHDL "wait" statements.

7. A method, performed by a data processing system having a memory of building a

digital circuit representation including a pipeline in the memory from a textual description of a loop, comprising the steps of:

identifying a loop carry dependency in said loop;

identifying a producer operation of said loop carry dependency;

identifying a consumer operation of said loop carry dependency;

determining a number, n, of cycles within which said producer operation must be scheduled after said consumer operation;

instantiating a placeholder node in said memory;

node-locking said placeholder node so that it must be scheduled n cycles after said consumer operation; and

constraining said producer operation to be scheduled before said placeholder node.

8. The method of claim 7, wherein the step of node-locking said placeholder node further comprises the step of creating a template structure in said memory which includes said placeholder node and said consumer operation.

9. The method of claim 8,

wherein said producer operation is included in a second template structure in said memory, and

wherein the step of constraining said producer operation further comprises the step of constraining said second template structure to be scheduled before said template structure.

10. The method of claim 7, wherein n is equal to an initiation interval of said pipeline multiplied by a number of iterations of said loop which execute before data produced by said producer is consumed by said consumer.

11. A method, performed by a data processing system having a memory, of building

a digital circuit representation in said memory, said digital circuit representation including a pipeline derived from a textual description of a loop, said method comprising the steps of:

identifying an access dependency of said loop;

identifying a first access operation of said access dependency;

identifying a second access operation of said access dependency;

determining a number, n, of cycles within which said second access operation must be scheduled after said first access operation;

instantiating a placeholder node in said memory;

node-locking said placeholder node so that it must be scheduled n cycles after said first access operation; and

constraining a scheduling order of said second access operation and said placeholder node.

12. The method of claim 11,

wherein said first access operation is chosen from the group of access operations including a memory read, a memory write, a signal write and a port write,

said second access operation is chosen from the group of access operations including a memory read, a memory write, a signal read, a signal write, a port read and a port write, and

the step of constraining said scheduling order of said second access operation and said placeholder node further includes the step of forcing said second access operation to be scheduled before said placeholder node.

13. The method of claim 11,

wherein said first access operation is chosen from the group of access operations including a memory read, a memory write, a signal read, a signal write, a port read and a port write,

said second access operation is chosen from the group of access operations

including a memory read, a memory write, a signal write and a port write, and

the step of constraining said scheduling order of said second access operation and said placeholder node further includes the step of forcing said second access operation to be scheduled before said placeholder node.

14. The method of claim 11,

wherein said first access operation is chosen from the group of access operations including a signal read and a port read,

said second access operation is chosen from the group of access operations including a signal read and a port read, and

the step of constraining said scheduling order of said second access operation and said placeholder node further includes the step of forcing said second access operation to be scheduled simultaneous with, or before said placeholder node.

15. The method of claim 11, wherein the step of constraining said scheduling order of said second access operation and said placeholder node further includes the step of forcing said second access operation to be scheduled before said placeholder node.

16. The method of claim 11, wherein the step of node-locking said placeholder node further includes the step of creating a template which includes said placeholder node and said first access operation.

17. The method of claim 11, wherein n is equal to an initiation interval of said pipeline multiplied by a number of iterations of said loop which execute between said first access operation and said second access operation.

18. A system for building, in a memory, a digital circuit representation which implements the behavior of a text description in said memory, said system having a processor coupled to a memory unit wherein said processor is programmed to perform

logic processing, said system comprising:

parsing logic for parsing said text description into a parsed text description, said text description including a loop with a delayed signal assignment having a delay value;

translating logic for translating said parsed text description into said digital circuit representation, said digital circuit including a pipeline; and

latency setting logic for setting a latency value of said pipeline to be said delay value of said delayed signal assignment.

19. A system as described in claim 18, wherein said pipeline implements said loop.

20. A system as described in claim 19, wherein said loop further includes a number, n, of wait statements, said system further comprising initiation interval setting logic for setting an initiation interval of said pipeline to be equal to n.

21. A computer program product comprising:

a computer usable medium having computer readable code embodied therein for building a digital circuit representation from a text description of a digital circuit, the computer program product comprising:

computer readable program code devices configured to cause a computer to effect parsing said text description, said text description including a loop with a delayed signal assignment having a delay value;

computer readable program code devices configured to cause a computer to effect translating said text description into said digital circuit representation including a pipeline; and

computer readable program code devices configured to cause a computer to effect setting a latency of said pipeline equal to said delay value.

22. The computer program product of claim 21 wherein said loop further includes N wait statements, where N is greater than zero, said computer program product further

comprising computer readable program code devices configured to cause a computer to effect setting an initiation interval of said pipeline equal to N.

23. A method performed by a data processing system having a memory, comprising the steps of:

parsing a text description of a circuit, said text description stored in the memory, said text description including a loop with N wait statements, where N is greater than zero;

translating said text description into a digital circuit representation in said memory, said digital circuit representation including a pipeline; and

setting an initiation interval of said pipeline equal to N.

24. The method of claim 23, wherein the wait statements are VHDL wait statements.

25. The method of claim 23, wherein the wait statements are Verilog HDL @posedge statements.

26. The method of claim 23, wherein the wait statements are Verilog HDL @negedge statements.

27. A system for building, in a memory, a digital circuit representation which implements the behavior of a text description in said memory, said system having a processor coupled to a memory unit wherein said processor is programmed to perform logic processing, said system comprising:

parsing logic for parsing said text description into a parsed text description, said text description including a loop with N wait statements, where N is greater than zero;

translating logic for translating said parsed text description into said digital circuit representation, said digital circuit including a pipeline; and

initiation interval setting logic for setting an initiation interval of said pipeline equal to N.

28. The system of claim 27, wherein the wait statements are VHDL wait statements.

29. The system of claim 27, wherein the wait statements are Verilog HDL @posedge statements.

30. The system of claim 27, wherein the wait statements are Verilog HDL @negedge statements.

31. A computer program product comprising a computer usable medium having computer readable code embodied therein for building a digital circuit representation from a text description of a digital circuit, the computer program product comprising:

computer readable program code devices configured to cause a computer to effect parsing said text description, said text description including a loop with N wait statements, where N is greater than zero;

computer readable program code devices configured to cause a computer to effect translating said text description into said digital circuit representation including a pipeline; and

computer readable program code devices configured to cause a computer to effect setting an initiation interval of said pipeline equal to N.

32. The method of claim 31, wherein the wait statements are VHDL wait statements.

33. The method of claim 31, wherein the wait statements are Verilog HDL

@posedge statements.

34. The method of claim 31, wherein the wait statements are Verilog HDL @negedge statements.

Abstract

METHODS FOR AUTOMATICALLY PIPELINING LOOPS

A method and an apparatus for creating a representation of a circuit with a pipelined loop from an HDL source code description. It infers a circuit including a pipelined loop which has cycle level simulation behavior matching that of the source HDL. Loop carry dependencies and memory and signal I/O accesses within the loop are scheduled correctly.

THIS PAGE BLANK (USPTO)

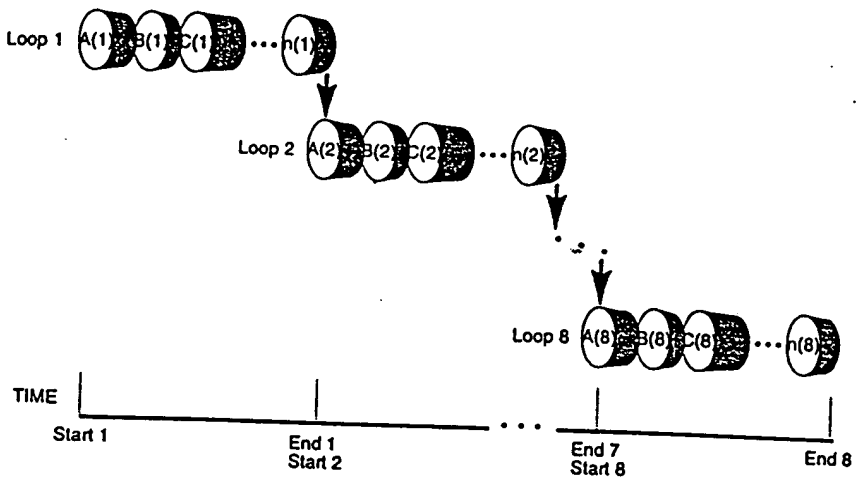

Loop 1
A(1)
B(1)
C(1)
n(1)
Loop 2
A(2)
B(2)
C(2)
n(2)
Loop 8
A(8)
B(8)
C(8)
n(8)
TIME
Start 1
End 1
Start 2
End 7
Start 8
End 8

Figure 1

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 1 of 26

THIS PAGE BLANK (USPTO)

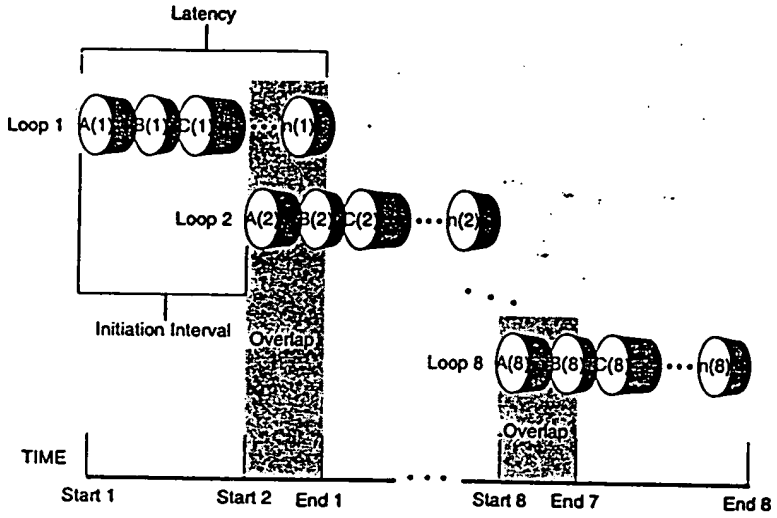

Latency
Loop 1
A(1)
B(1)
C(1)
h(1)
Loop 2
A(2)
B(2)
C(2)
h(2)
Initiation Interval
Overlap
Loop 8
A(8)
B(8)
C(8)
h(8)
Overlap
TIME
Start 1
Start 2
End 1
Start 8
End 7
End 8

Figure 2

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 2 of 26

THIS PAGE BLANK (USPTO)

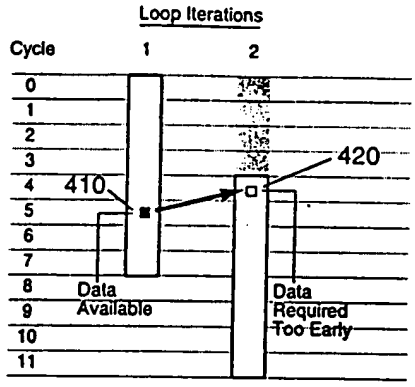


Figure 3 (a)

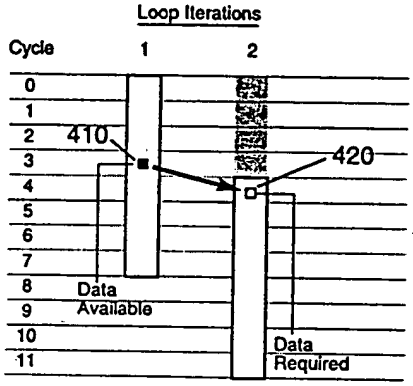


Figure 3 (b)

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 3 of 26

THIS PAGE BLANK (USPTO)

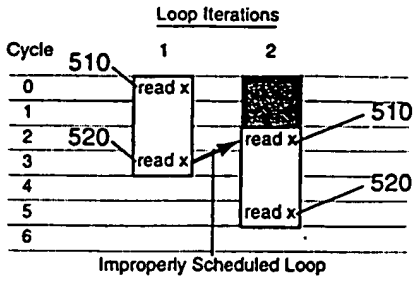


Figure 4 (a)

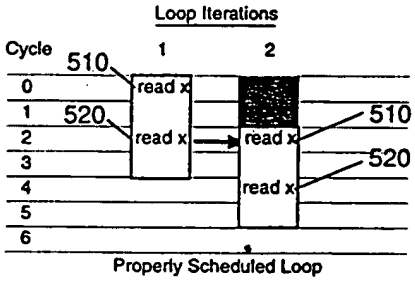


Figure 4 (b)

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USPTO)

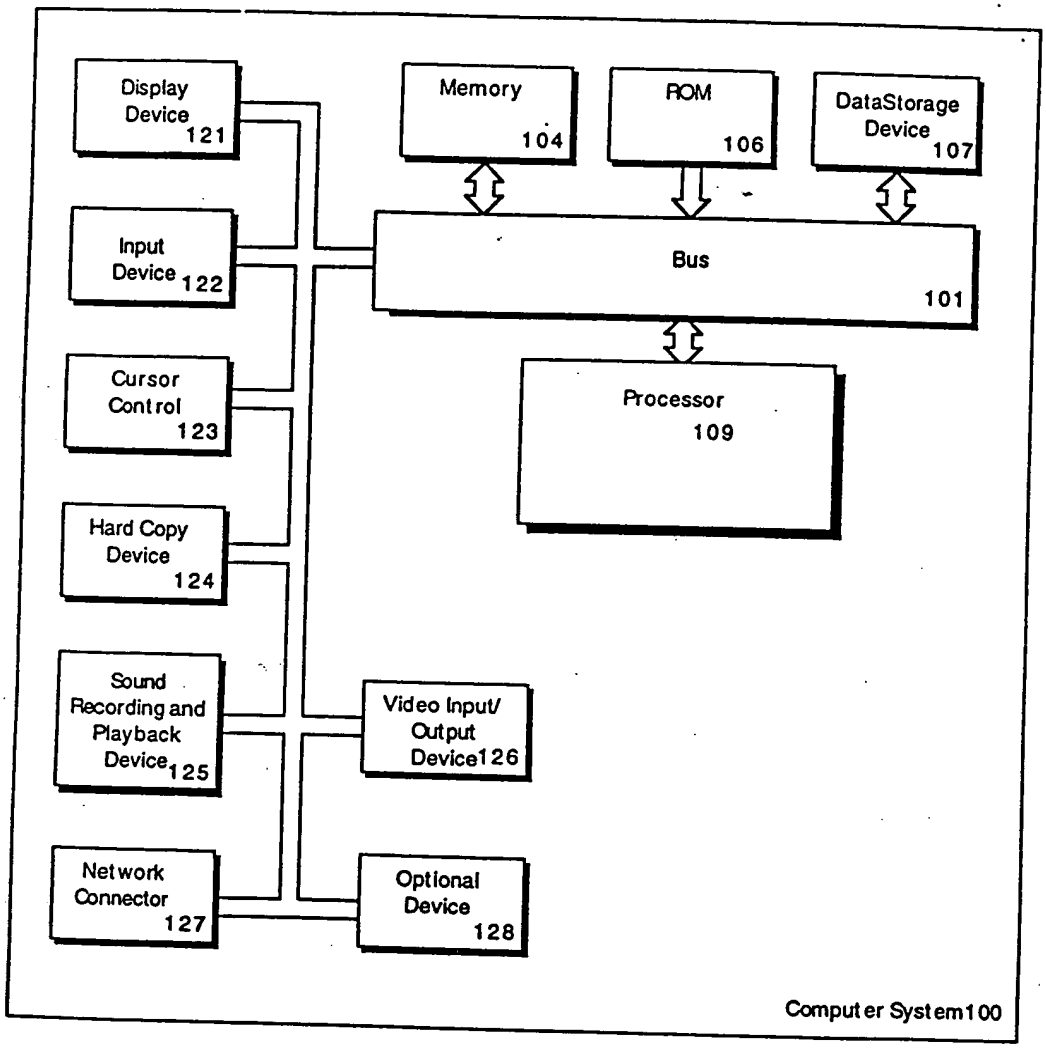

Display Device 121
Memory 104
ROM 106
DataStorage Device 107
Input Device 122
Bus 101
Cursor Control 123
Processor 109
Hard Copy Device 124
Sound Recording and Playback Device 125
Video Input/ Output Device126
Network Connector 127
Optional Device 128
Computer System100

Figure 5

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USPTO)

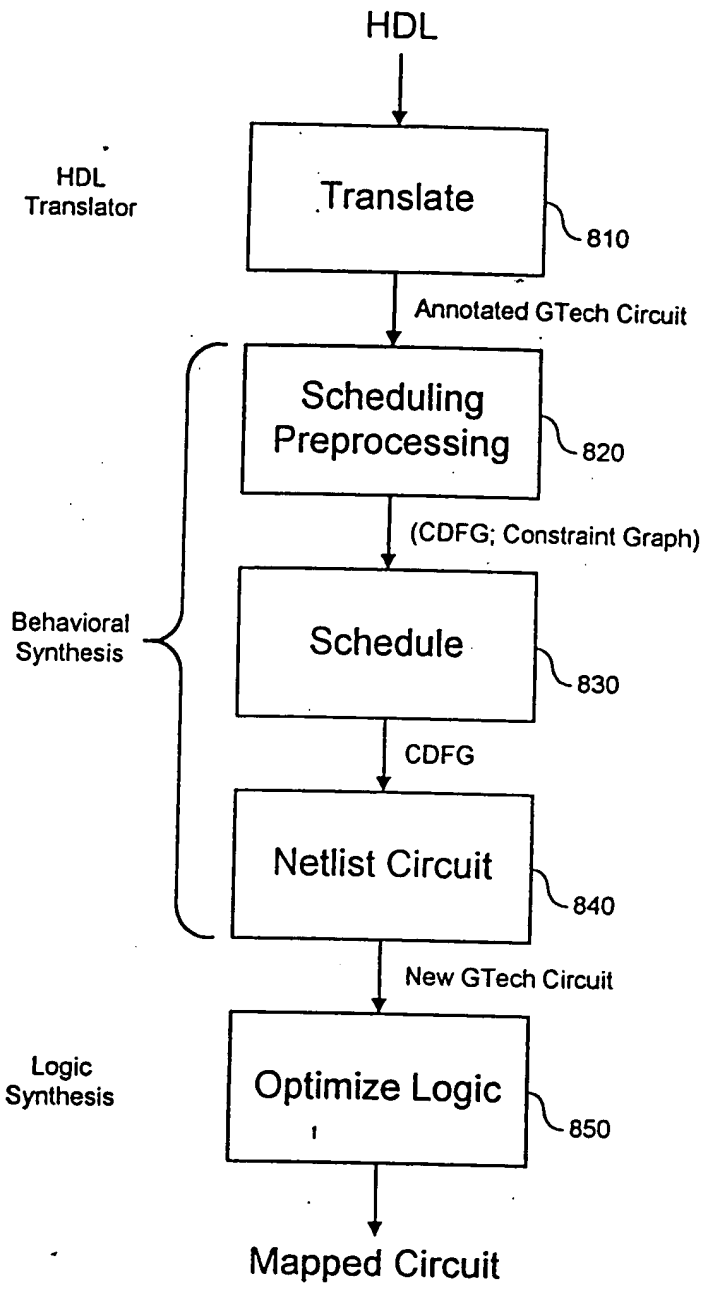


Figure 6 Synthesis with Scheduling

Atty. Docket No. 4000/...
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 6 of 26

THIS PAGE BLANK (USPTO)

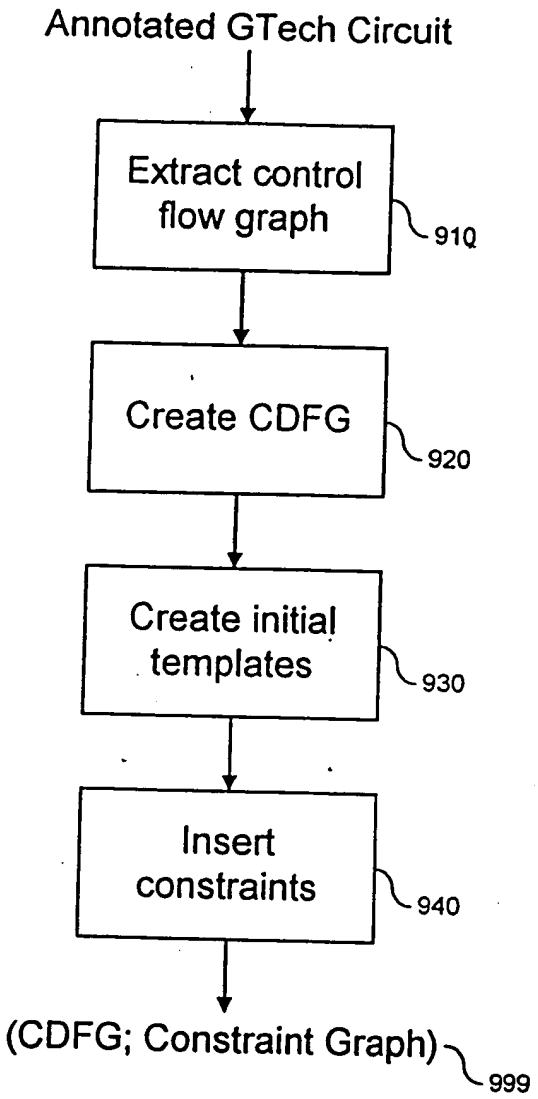


Figure 7

Scheduling Preprocessing

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 7 of 26

THIS PAGE BLANK (USPTO)

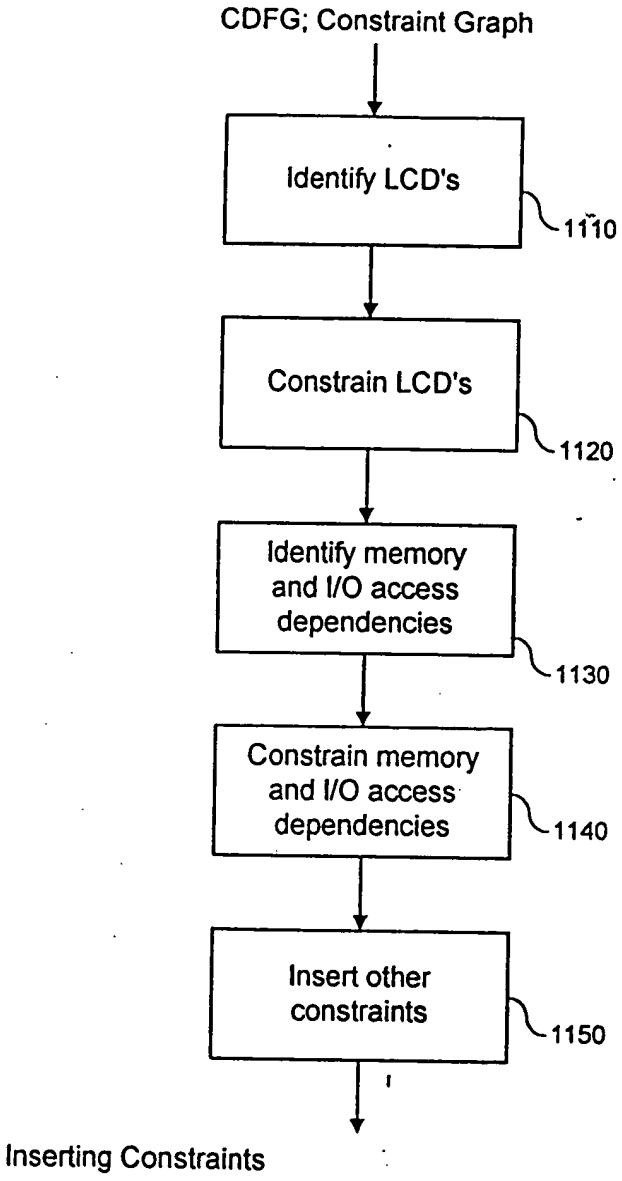

CDFG; Constraint Graph
Identify LCD's
1110
Constrain LCD's
1120
Identify memory and I/O access dependencies
1130
Constrain memory and I/O access dependencies
1140
Insert other constraints
1150
Inserting Constraints

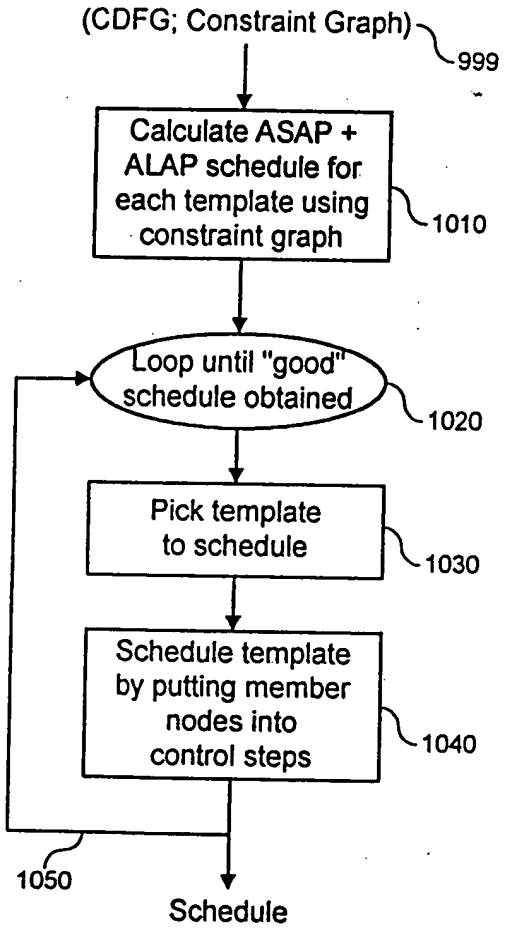
Figure 8

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 8 of 26

THIS PAGE BLANK (USPTO)



Scheduling Using Templates

Figure 9

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USP

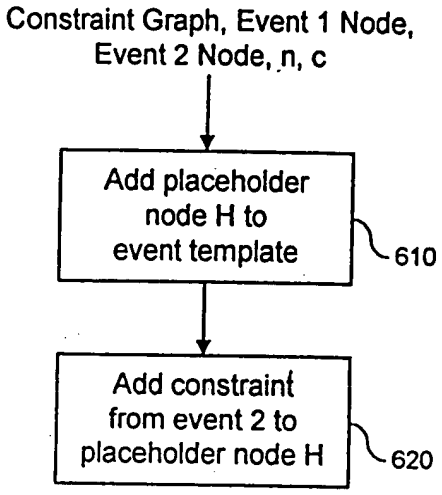

Constraint Graph, Event 1 Node, Event 2 Node, n, c
Add placeholder node H to event template
610
Add constraint from event 2 to placeholder node H
620

Figure 10

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 10 of 26

THIS PAGE BLANK (USPTO)

```
module loopex8 ( c, x, y, z, clock);
input [1:0] x, y, z;
   input clock ;
   output [2:0] c;
   reg [2:0] c;
   reg [2:0] p;

   always begin                                    3030

      forever begin : theloop                              3010

            c <= x - p ;

            @(posedge clock) ;                             3020

            p = y + z ;

            @(posedge clock) ;

      end

   end

endmodule
```

Figure 11

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 11 of 26

THIS PAGE BLANK (USPTO)

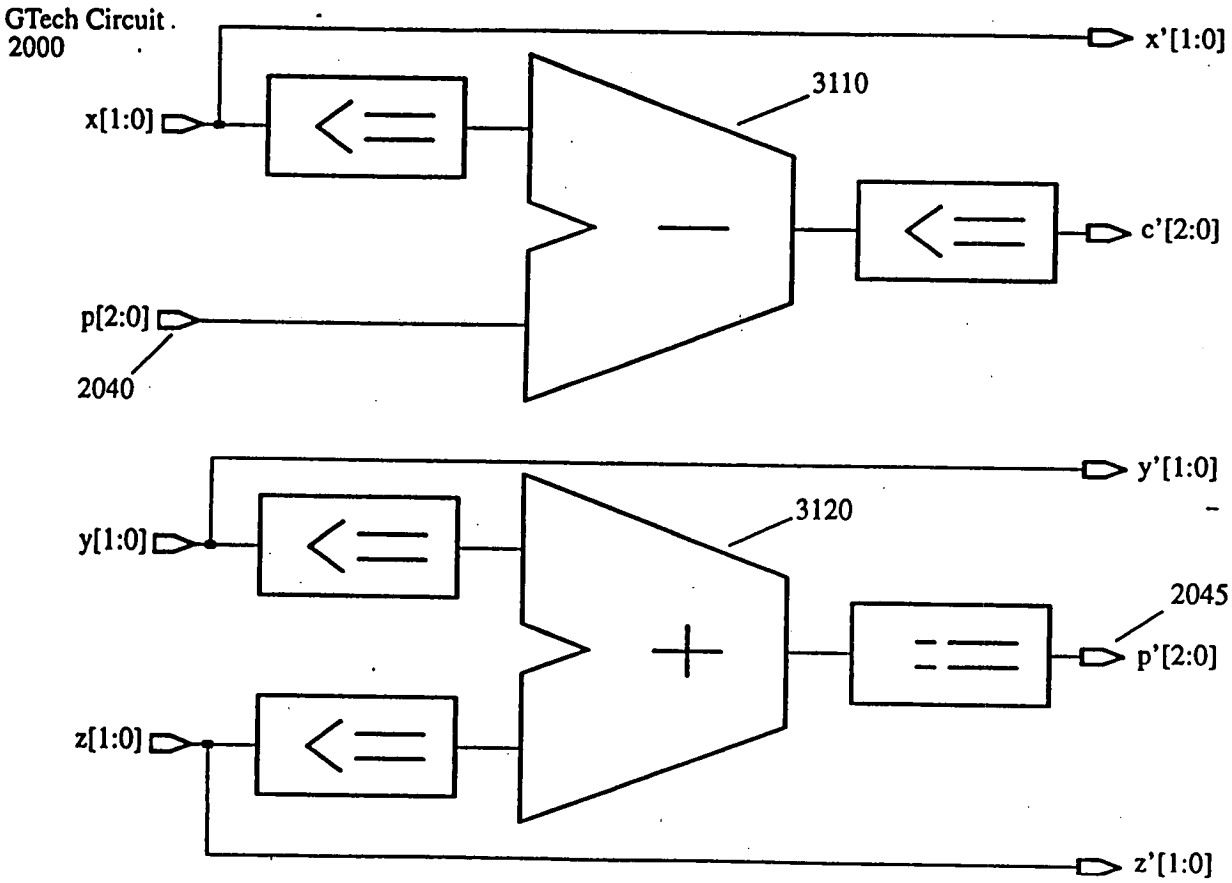

GTech Circuit
2000
x'[1:0]
x[1:0]
3110
c'[2:0]
p[2:0]
2040
y'[1:0]
y[1:0]
3120
2045
p'[2:0]
z[1:0]
z'[1:0]

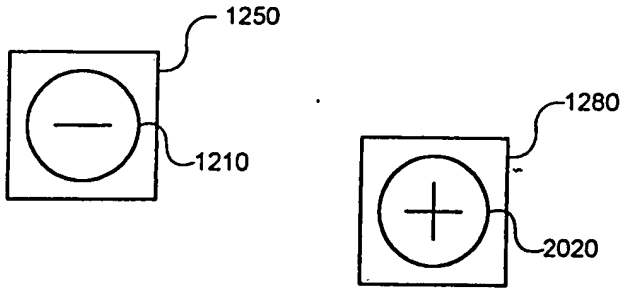
Figure 12

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 12 of 26

THIS PAGE BLANK (USPTO)



n = 2

Figure 13a

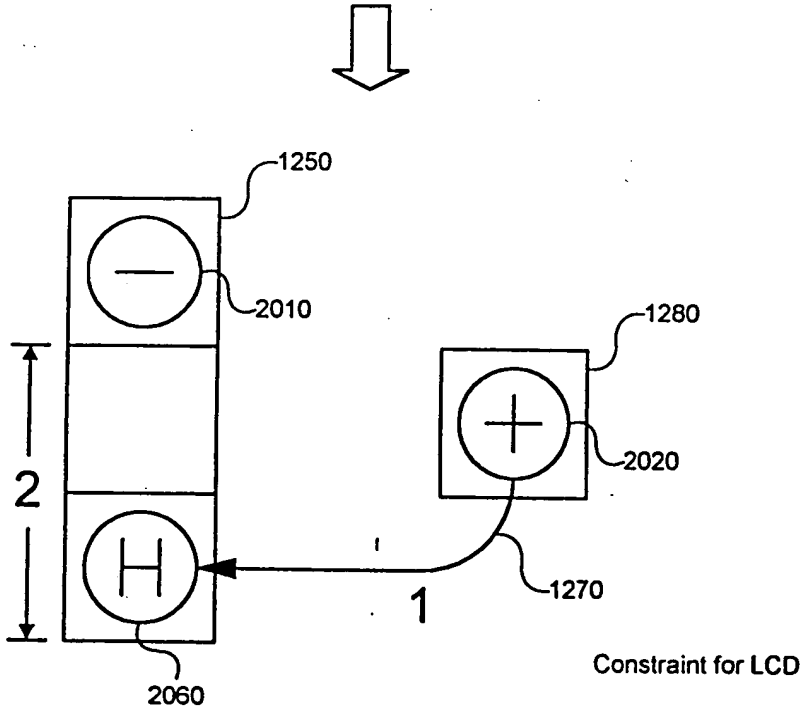


Figure 13b

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 13 of 26

THIS PAGE BLANK (USPTO)

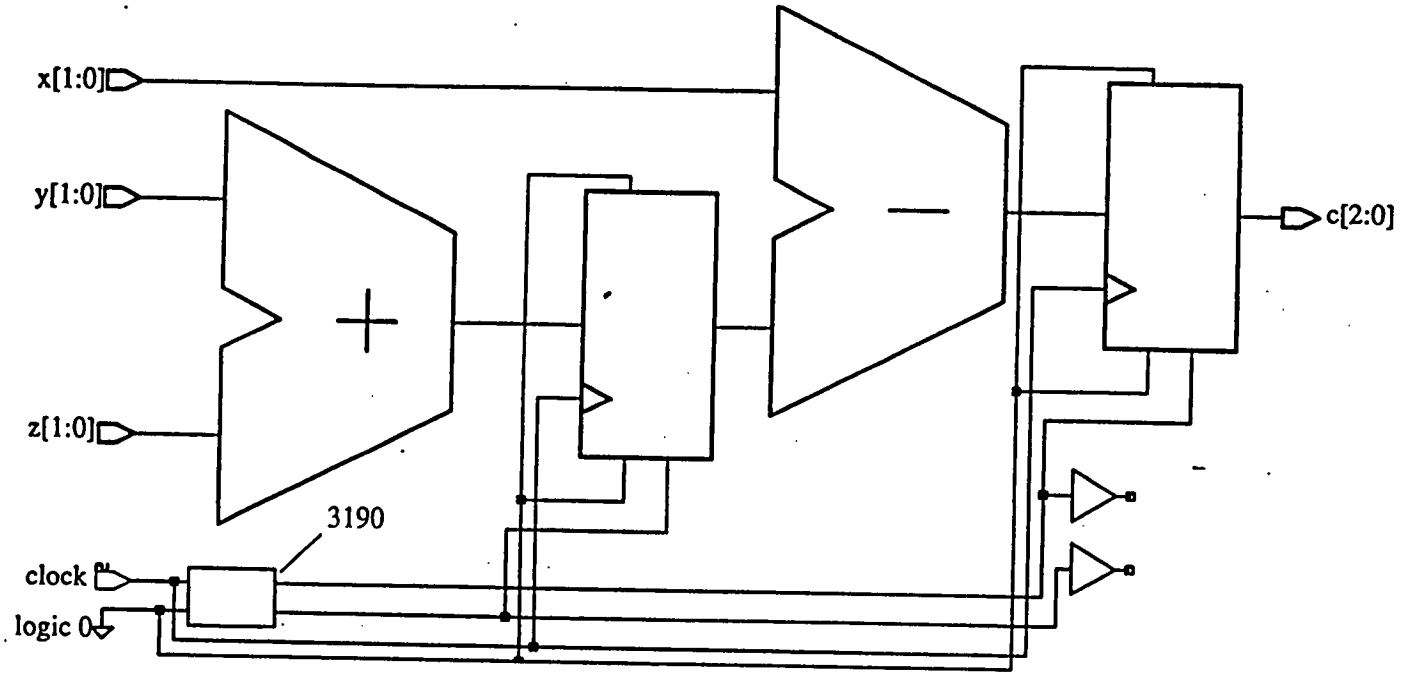


Figure 14

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 14 of 26

THIS PAGE BLANK (USPTO)

```
module write4 ( w, x, clock);

  input [15:0] x ;
  input clock ;
  output [31:0] w;
  reg [32:0] w;
  reg [15:0] x1 ;
  reg [15:0] x2 ;

  always begin                          1530

      forever begin : writeloop         1530

            x1 <= x ;

            @(posedge clock) ;          1530

            x2 <= x ;

            w <= x1 * x2 ;

      end

  end

endmodule
```

Figure 15

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USPTO)

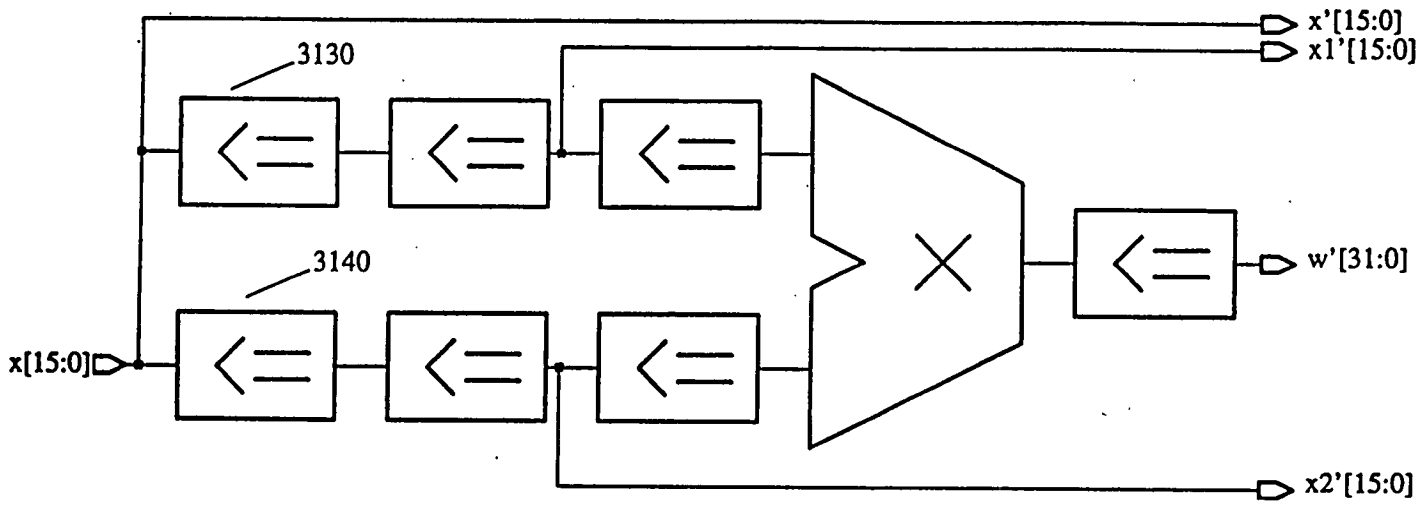


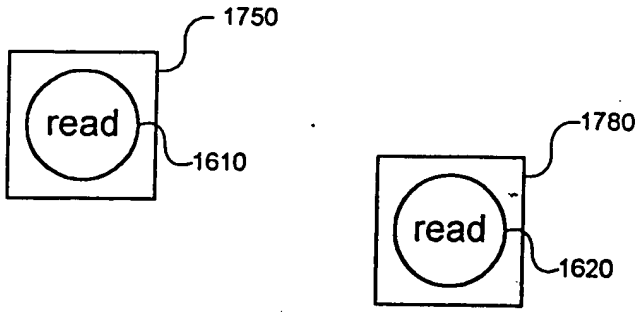
Figure 16

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 16 of 26

THIS PAGE BLANK (USPTO)



n = 1

Figure 17a

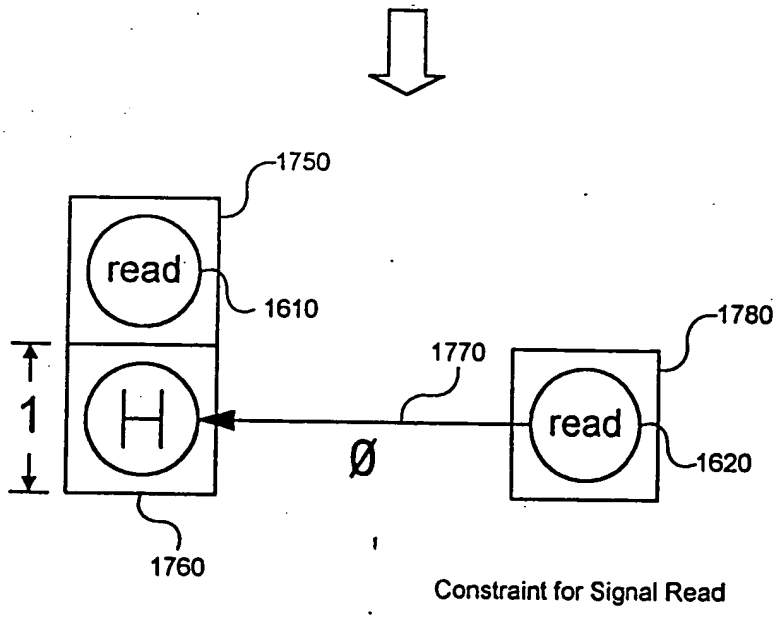


Figure 17b

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 17 of 26

THIS PAGE BLANK (USPTO)

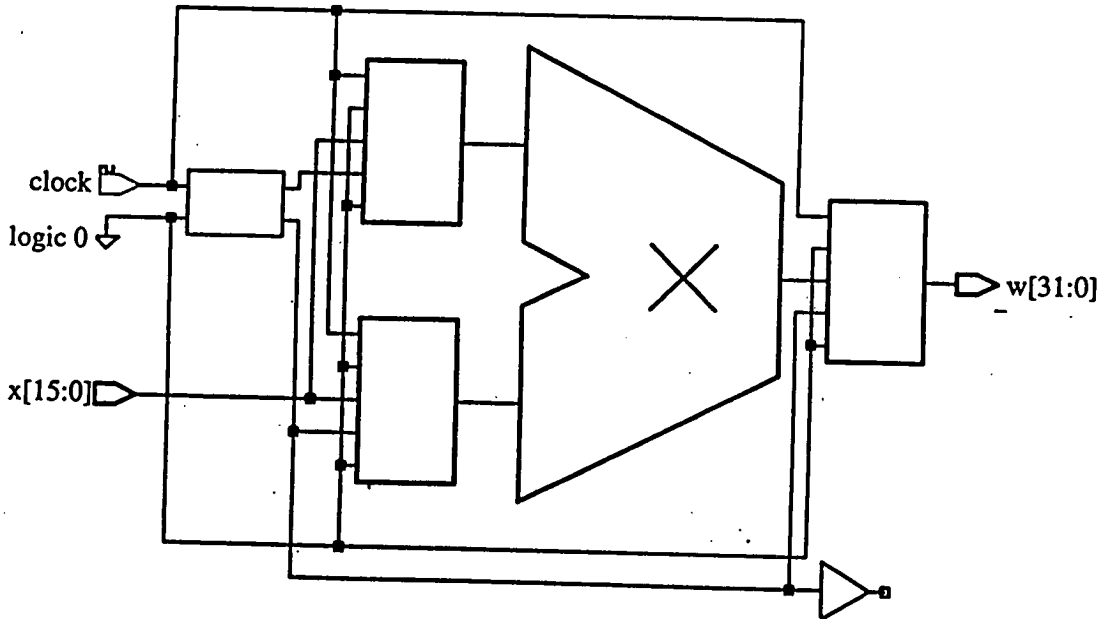


Figure 18

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 18 of 26

THIS PAGE BLANK (USPTO)

```
module after1 ( c, x, y, z, clock);

  input [1:0] x, y, z;
  input clock ;
  output [2:0] c;
  reg [2:0] c;
  reg [2:0] p;

  always begin

        @(posedge clock) ;

        forever begin
              c <= #24 x - p ;

              @(posedge clock) ;

              p = y + z ;

              @(posedge clock) ;
        end
  end
endmodule
```

Figure 19 (a)

```
entity after1 is
  port(
        c : out integer range 0 to 7;
        x, y, z : in integer range 0 to 3;
        clock : in bit
  );
end after1;

architecture behavioral of after1 is begin
  process
        variable p : integer range 0 to 7;
  begin
        wait until clock'event and clock = '1';

        loop

          c <= transport x - p after 24 ns;

          wait until clock'event and clock = '1';

          p := y + z;

          wait until clock'event and clock = '1';

        end loop;

  end process;
end behavioral;
```

Figure 19 (b)

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USPTO)

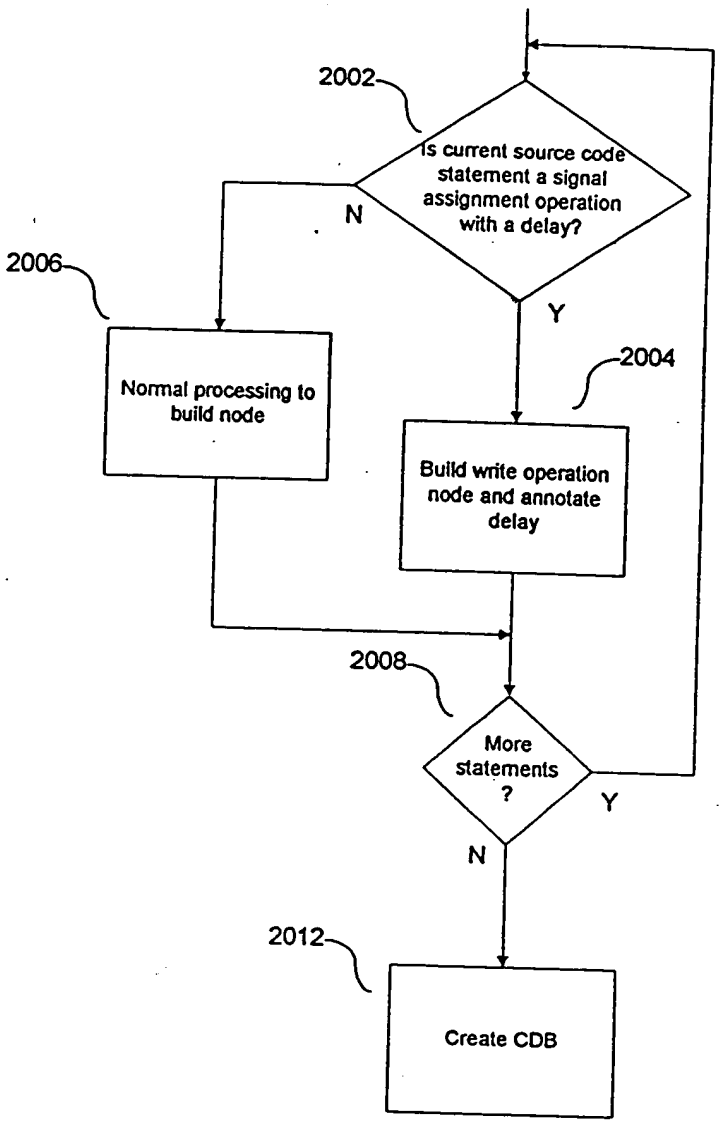

2002
Is current source code statement a signal assignment operation with a delay?
N
Y
2006
Normal processing to build node
2004
Build write operation node and annotate delay
2008
More statements ?
Y
N
2012
Create CDB

Fig. 20

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 20 of 26

THIS PAGE BLANK (USPTO)

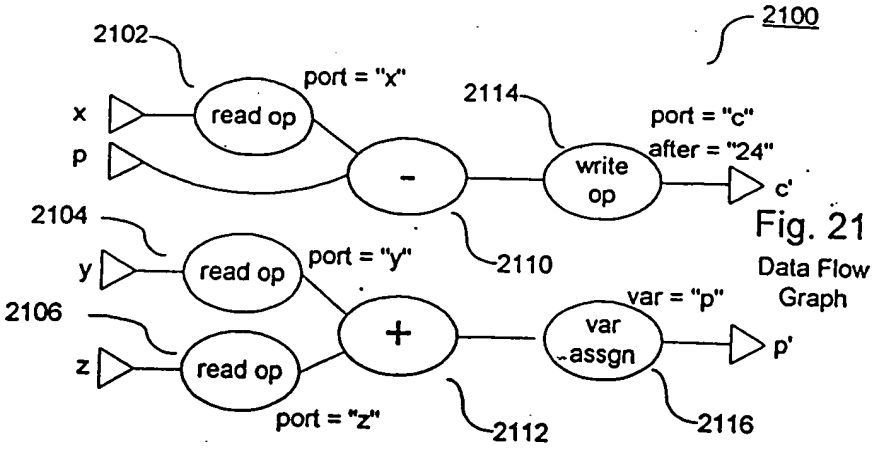


Fig. 21
Data Flow Graph

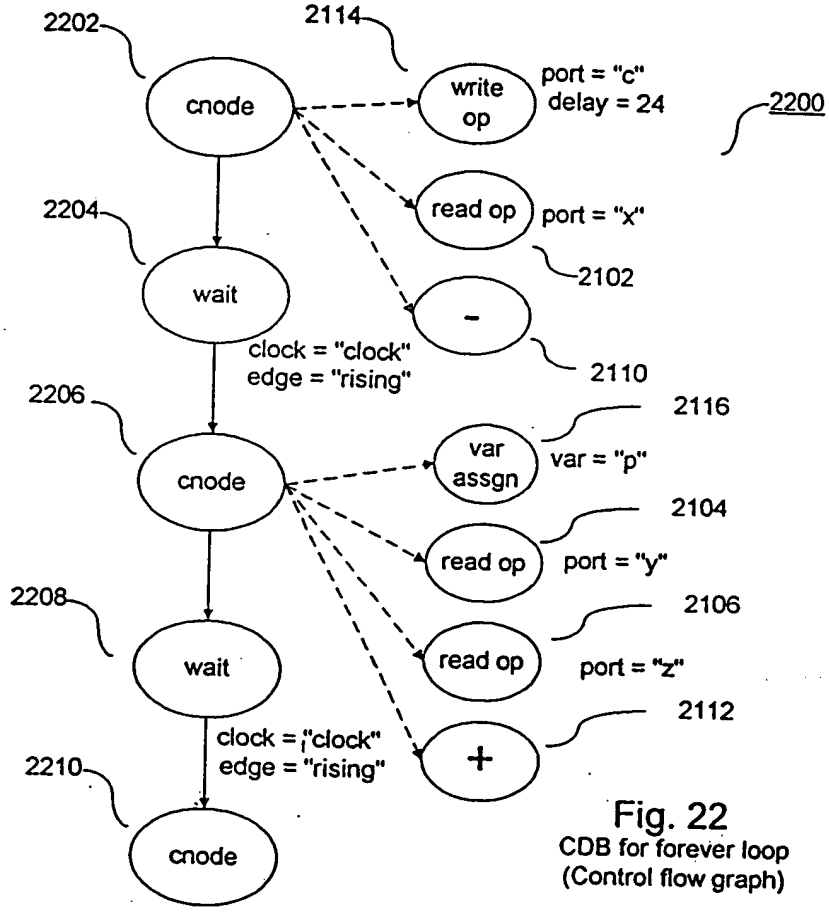


Fig. 22
CDB for forever loop
(Control flow graph)

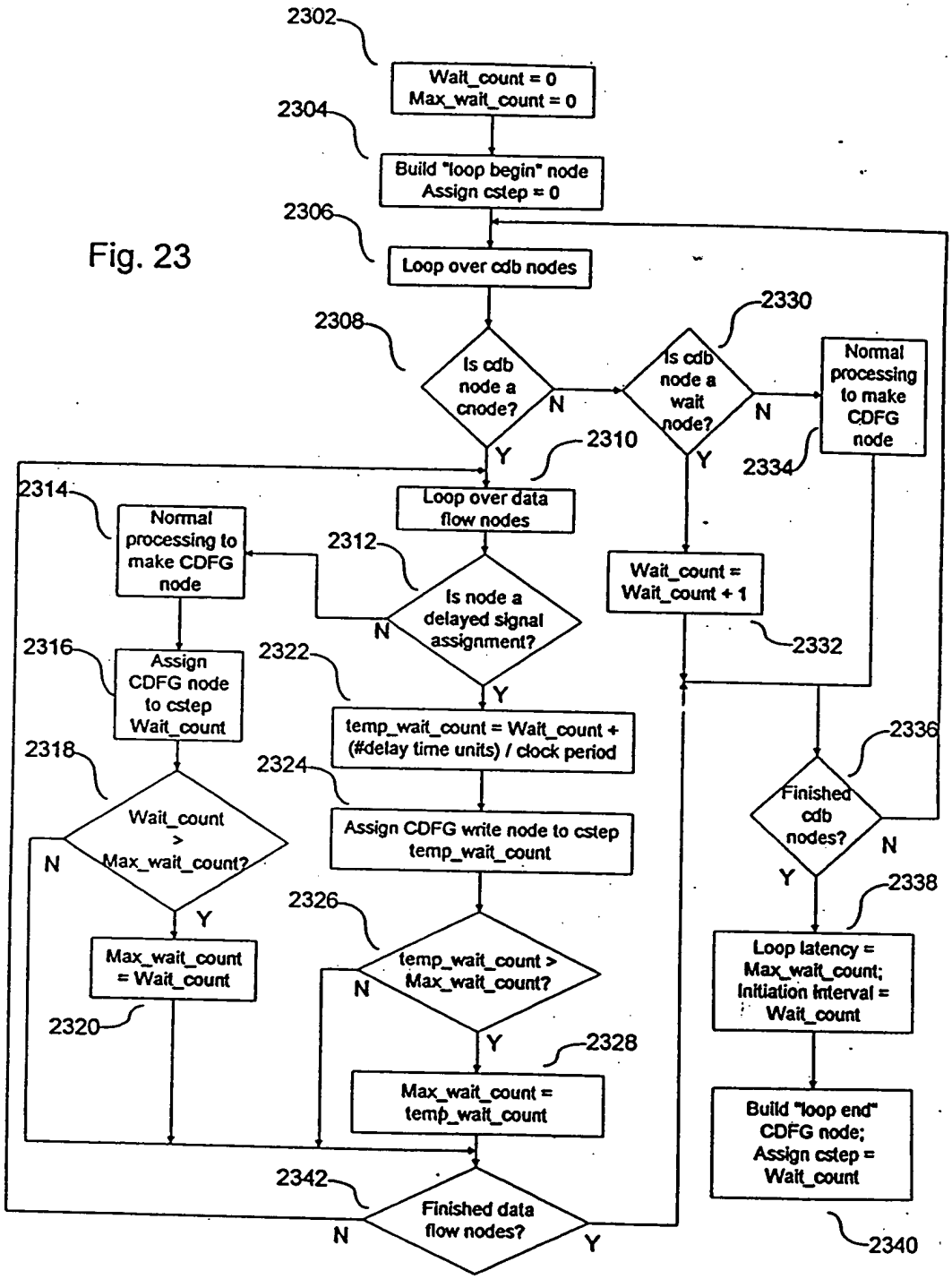
Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 21 of 26

THIS PAGE BLANK (USPTO)

Fig. 23


2302
Wait_count = 0
Max_wait_count = 0
2304
Build "loop begin" node
Assign cstep = 0
2306
Loop over cdb nodes
2308
Is cdb node a cnode?
N
Y
2330
Is cdb node a wait node?
N
Y
Normal processing to make CDFG node
2334
2310
Loop over data flow nodes
2314
Normal processing to make CDFG node
2312
Is node a delayed signal assignment?
N
Y
Wait_count = Wait_count + 1
2332
2316
Assign CDFG node to cstep Wait_count
2322
temp_wait_count = Wait_count + (#delay time units) / clock period
2318
Wait_count > Max_wait_count?
N
Y
2324
Assign CDFG write node to cstep temp_wait_count
Max_wait_count = Wait_count
2320
2326
temp_wait_count > Max_wait_count?
N
Y
2328
Max_wait_count = temp_wait_count
2336
Finished cdb nodes?
N
Y
2338
Loop latency = Max_wait_count; Initiation interval = Wait_count
Build "loop end" CDFG node; Assign cstep = Wait_count
2340
2342
Finished data flow nodes?
N
Y

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USPTO)

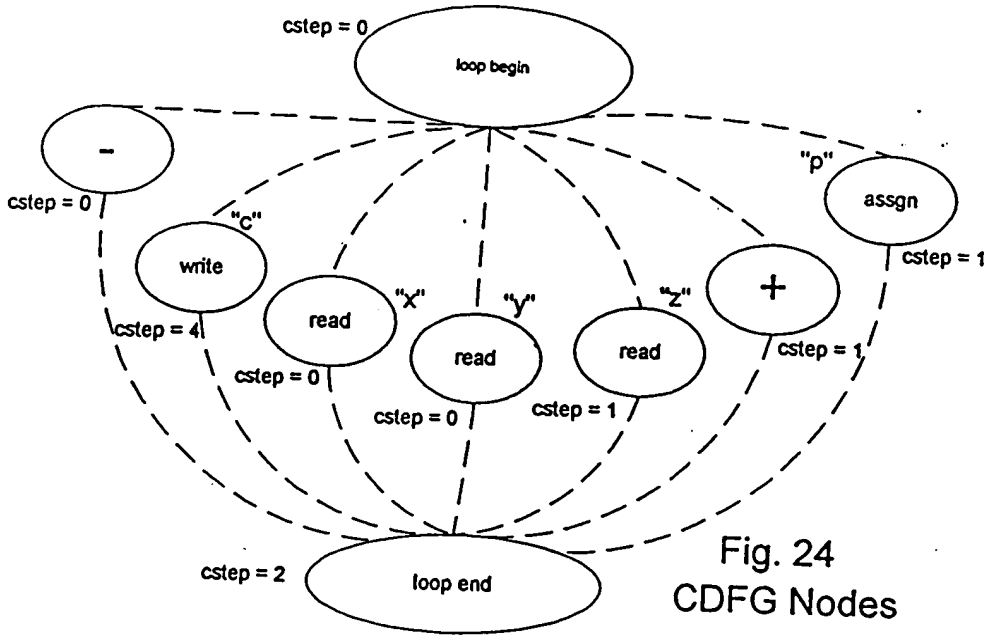


Fig. 24
CDFG Nodes

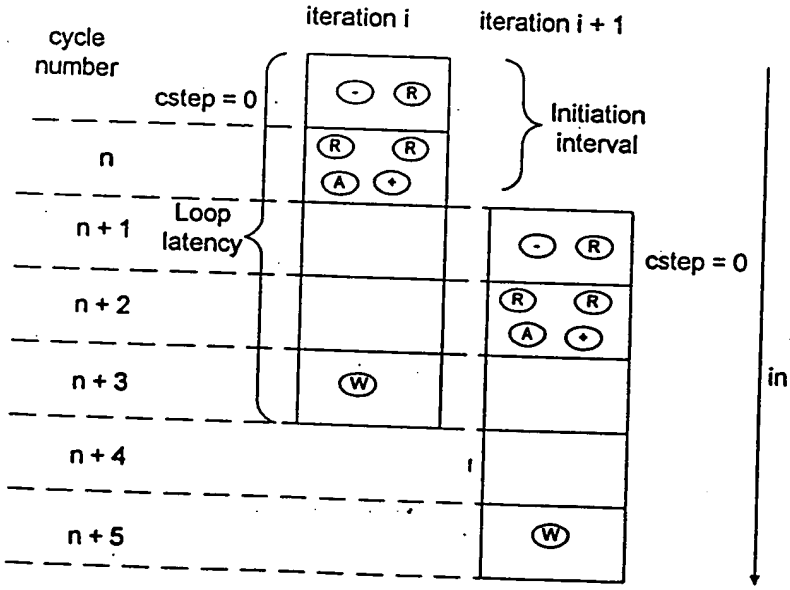


Fig. 26

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USPTO)

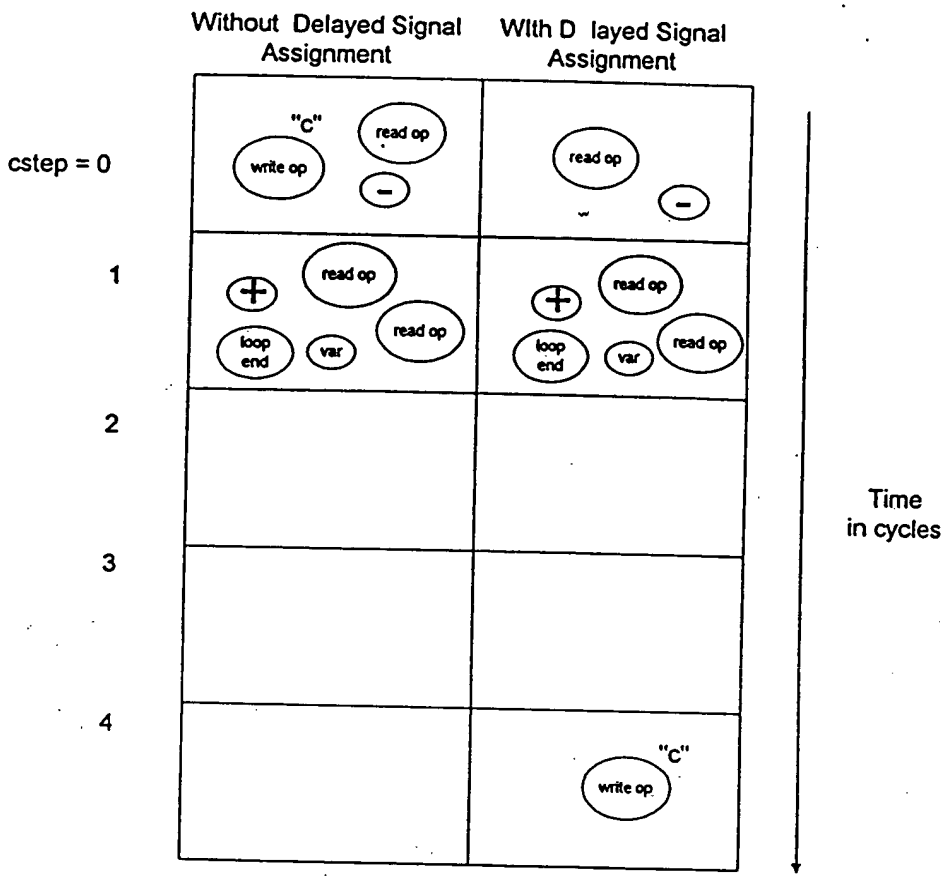

Without Delayed Signal Assignment
With D layed Signal Assignment
cstep = 0
1
2
3
4
"C"
read op
write op
read op
read op
loop end
var
read op
read op
loop end
var
read op
"C"
write op
Time in cycles

Fig. 25

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USPTO)

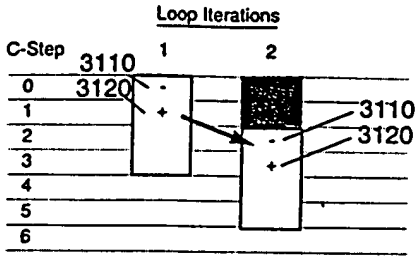


Figure 27

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 25 of 26

THIS PAGE BLANK (USPTO)

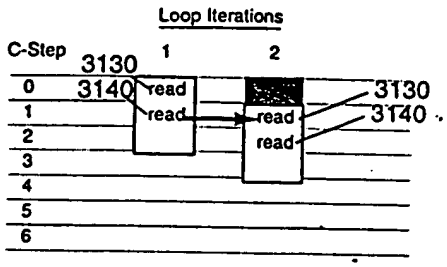

Loop Iterations
C-Step
1
2
3130
3140
read
read
read
read
3130
3140
0
1
2
3
4
5
6

Figure 28

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

Main Reissue Application Figures
Sheet 26 of 26

THIS PAGE BLANK (USPTO)

Appendix A

THIS PAGE BLANK (USPTO)

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX A
Sheet 1 of 7

THIS PAGE BLANK (USPTO)

Scheduling using Behavioral Templates

Tai Ly, David Knapp, Ron Miller, Don MacMillen
Synopsys Inc.
700B E. Middlefield Road
Mountain View, CA USA 94043

Abstract: This paper presents the idea of "behavioral templates" in scheduling. A behavioral template locks several operations into a relative schedule with respect to one another. This simple construct proves powerful in addressing: (1) timing constraints, (2) sequential operation modeling, (3) pre-chaining of certain operations, and (4) hierarchical scheduling. We present design examples from industry to demonstrate the importance of these issues in scheduling.

1.0 Introduction

The task of scheduling [4] is to sequence nodes in a control and data flow graph (CDFG) by assigning each node to a control step (*cstep*). We present the idea of behavioral templates, and describe how we use behavioral templates to address several issues that arise when applying scheduling to commercial designs. For the purpose of this paper, we assume timing constrained scheduling [5].

A behavioral template specifies a relative scheduling among its member CDFG nodes. It is a template in the sense that its member nodes can be treated as a single scheduling unit by assigning the starting cstep for the template. It is behavioral in the sense that it specifies a scheduling pattern as opposed to, for example, a structural pattern [8]. We extend scheduling algorithms to handle behavioral templates by recasting the task of scheduling as that of assigning templates to csteps.

Although a simple idea, behavioral templates provide a powerful way to address four issues in scheduling:

1. **Timing constraints.** We use behavioral templates to impose fixed and maximum timing constraints. This is more efficient than using precedence edges alone because an entire sequence of nodes is considered at once when scheduling one template.
2. **Multi-cycle operations.** To enable scheduling of complex multi-cycle operations, we use multiple CDFG nodes locked in a behavioral template to model the cycle-by-cycle I/O and resource requirements of such operations.
3. **Logic and bit-manipulation operations.** We use behavioral templates to force certain chaining of logic and bit-manipulation operations to save register costs. This reduces the scheduling design space, and therefore run times.
4. **CDFG hierarchy.** We implement hierarchical scheduling by inlining each scheduled subgraph, using a behavioral template to lock the inlined nodes according to the subgraph's schedule.

This paper is organized as follows. Section 2 compares this work to previous research. Section 3 defines behavioral templates. Section 4 describes extending scheduling for behavioral templates. Secti n 5 discusses applications. Section 6 presents results. Section 7 concludes this paper.

2.0 Related Work

The term "template" was used in [8] to describe structural patterns to exploit regularity. In [9] and [10], such templates are used t guide the clustering of CDFG nodes into super nodes which map to "regular" subcircuits. Both of these works focus on extracting regular patterns by pattern matching, whereas our work focuses on how to schedule a set of behavioral patterns. Our behavioral templates do not represent *repeating* patterns, but specify local scheduling constraints among CDFG nodes.

Most scheduling systems model multi-cycle operations using single CDFG nodes whose delays are greater than 1. In [7], multi-cycle operations are treated as multiple single-cycle operations. This turns out to be similar to our template-based model for sequential operations, except that we make deliberate use of cycle-by-cycle input/output and resource requirements to model complex operations.

Hierarchical scheduling based on super nodes are used in [9], [6], and [7]. We know of no other system which hierarchically schedules a design while taking advantage of possible resource sharing between nodes and edges in different subgraphs.

3.0 Behavioral Templates

We define a behavioral template, T, as a CDFG object which specifies a set of tuples, (n_i, o_i), where n_i is a CDFG node and o_i is an integer cycle *offset*. The semantics is that T imposes the constraint:

$$\text{schedule}(n_i) = \text{schedule}(T) + o_i \qquad \text{for all } (n_i, o_i) \text{ in } T$$

where schedule(n_i) and schedule(T) denotes the schedules for n_i and T, respectively.

That is, if T is scheduled to cstep j, then every member node, n_i, f T must be scheduled to the cstep, $j + o_i$. This locks all nodes in T into a pattern f relative schedules, and we may schedule th entire group of nodes by scheduling the template T itself. Fig. 1(a) shows a template, T1 = { (a,0) (b,1) (c,2) (d,3) (e,5) }, containing 5 nodes. All CDFG edges have been omitted f r clarity. (In the figures, we show behavioral template as a box containing one or more nodes in

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX A
Sheet 2 of 7

THIS PAGE BLANK (USPTO)

slots. The top slot in the box is offset 0, the second slot from top is offset 1, and so on. For example, the node "e" in Fig. 1(a) has offset 5 in T1 because it is in the 6th slot from the top of the box.)

Whenever a node is a member of two or more different templates, we can always merge these templates into one. Consider the templates T1 and T2 in Fig. 1(a) and 1(b). If node g is to be added to template T1 at offset 1, then we merge T1 and T2 into T3 of Fig. 1(c).

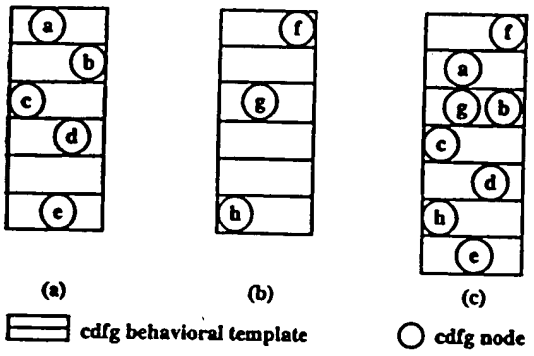


FIGURE 1. Template examples (a) T1 = { (a,0) (b,1) (c,2) (d,3) (e,5) }; (b) T2 = { (f,0) (g,2) (h,5) }; (c) T3 = { (f,0) (a,1) (g,2) (b,2) (c,3) (d,4) (h,5) (e,6) }

4.0 Scheduling with Behavioral Templates

Instead of scheduling individual CDFG nodes, we restate the scheduling problem in terms of behavioral templates. Initially, we create one template for every CDFG node, and then merge templates whenever nodes are added to other templates. This ensures that every CDFG node is a member of one and only one template. The timing constrained scheduling task is then to schedule all templates to minimize resource costs subject to timing constraints between templates. This section describes how we extend existing scheduling algorithms for behavioral templates.

4.1 Timing Constraints between Templates

From the CDFG, we construct a weighted, directed graph G=(V , E) where V is the set of all behavioral templates in the CDFG, and E is the set of directed edges between templates. The weight $d(T_x, T_y)$ of an edge $e(T_x , T_y)$ in E specifies the minimum delay between the schedules of T_x and T_y, i.e.,

$$\text{schedule}(T_x) + d(T_x , T_y) <= \text{schedule}(T_y) \qquad \text{Eq. 1}$$

The edges in E are constructed from the data/control dependencies between member nodes in the templates. For every pair of templates, T_x and T_y, $d(T_x , T_y)$ is the maximum value of

$$w(n_i , n_j) + o_i - o_j \qquad \text{Eq. 2}$$

over all (n_i , o_i) in T_x and all (n_j , o_j) in T_y, where $w(n_i , n_j)$ is the minimum cycle delay from node n_i to node n_j.

Note that Eq. 2 can be negative. This means $d(T_x , T_y)$ can be negative, and the graph G is not acyclic. If G contains any cycle of positive lengths, then the timing constraints are unsatisfiable. To check for positive cycles, we solve for the all-pairs-longest-path problem for G using a simple $O(N^3)$ algorithm, where N is the number of templates in G. The longest path lengths are stored in a matrix, LP, for subsequent incremental update of the as soon as possible (ASAP) and as late as possible (ALAP) schedules.

4.2 ASAP and ALAP Schedules

At the start of scheduling, we calculate the ASAP and ALAP schedules for all templates in G, to establish the scheduling time frame for each template. Since G may contain negative weighted edges, we use a relaxation algorithm similar to that in [3] to compute the initial ASAP/ALAP schedules:

(1) Propagate along positive edges in E only;

- for ASAP, propagate forward from the source of CDFG;
- for ALAP, propagate backward from the sink of CDFG;

(2) Relax schedules to satisfy constraints implied by negative edges in E;

(3) Repeat step 1 until no more changes in relaxation step.

When there are no positive cycles in G, the above algorithm is guaranteed to converge in e+1 iterations where e is the number of negative edges in E. The overall computational complexity is $O(N^2e)$ where N is number of templates in G.

The ASAP and ALAP schedules define the initial time frames. Subsequently, as each template is scheduled, we update the time frames of other templates using the longest path lengths matrix, LP:

$$\text{schedule}(T_x) + LP(T_x , T_y) <= \text{schedule}(T_y) \text{ for all } T_x , T_y \text{ in V}$$

There is no need for relaxation in this incremental update because LP already takes into account all negative edges in E.

4.3 Cost Functions

We use a number of iterative/constructive scheduling algorithms each of which successively picks an unscheduled template and schedules it to a cstep in its time frame. The algorithms differ in how they pick the next template to schedule, and in how they pick which cstep to schedule the template to. We define the template priority/cost functions in terms of priority/cost functions on the CDFG nodes.

For example, in our implementation of list scheduling, the template priority function is defined as the maximum of its member nodes' priority values. This gives priority to the template containing the highest priority nodes. In our implementation of greedy scheduling, the incremental cost function for scheduling a template $T=\{(n_i , o_i)\}$ to a cstep j, is defined as the sum total of the incremental costs for scheduling nodes n_i to csteps $j + o_i$.

Scheduling/de-scheduling moves on templates are implemented as moves on their member nodes. All data structures are updated as CDFG nodes are scheduled/de-scheduled. In particular, resource costs for functional units, registers and interconnects are still computed according to the lifetimes and mutual exclusivity of CDFG nodes and edges. This approach is easy to implement and leverages previous work on scheduling CDFG nodes.

4.4 Pre-assigned Operations

Allowing negative edges in G requires that we extend scheduling algorithms to handle maximum timing constraints. This is complicated by "pre-assigned" operations, i.e., operations that are

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX A
Sheet 3 of 7

THIS PAGE BLANK (USPTO)

assigned t specific resources before scheduling. Examples f pre-assigned operations are memory read/write perations for th same RAM. We use a list scheduling algorithm to find an initial legal schedule based on source code ordering. However, list scheduling can fail to find a legal schedule when there are maximum timing constraints. So we augment list scheduling with a recovery step. When list scheduling fails, the recovery step relaxes the template schedules that caused scheduling failures, and iterates:

1. List scheduling step:

Successively consider operations in the ready list in increasing source code ordering. For each ready operation, n_i, check its template, $T_x = \{...(n_i, o_i)...\}$, for scheduling in the cstep s - o_i, where s is the current cstep. Postpone scheduling of T_x if any of the following is true:

- T_x has a "relaxed cstep" (see step 2) which is greater than s - o_i
- T_x has no "relaxed cstep", but ASAP is greater than s - o_i
- there is a resource contention if T_x is scheduled to s - o_i

When all nodes have been scheduled, exit with success.

If T_x is postponed due to resource contention, and if s - o_i is greater than or equal to the ALAP cstep for T_x, then list scheduling has failed. When this happens, go to step 2 and try to recover.

2. Recovery step:

When list scheduling fails to find a legal schedule for T_x, we try to recover by increasing its ALAP cstep and rerun list scheduling in step 1. In order to increase the ALAP cstep for T_x, we find all scheduled templates, T_y, for which

$$alap(T_x) == schedule(T_y) - LP(T_x, T_y) \qquad \text{Eq. 3}$$

where $alap(T_x)$ is the ALAP cstep for T_x. For every such template, T_y, we set its "relaxed cstep" to $schedule(T_y) + 1$. This forces the next run of list scheduling to schedule T_y one cstep later.

This step exits with failure if any of the following is true:

- $ALAP(T_x)$ is at maximum global cstep
- there is no template T_y which satisfies Eq. 3
- algorithm has iterated for N times (N is the # of templates)

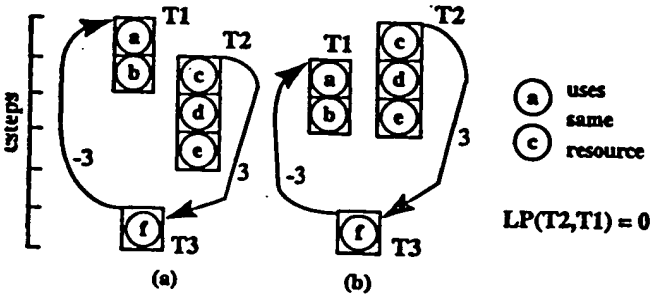


FIGURE 2. Example of list scheduling failure and recovery (a) first iteration fails at T2 (b) second iteration succeeds with T1 relaxed to cstep 1

Fig. 2 shows an xample of this algorithm at work. In this example, CDFG nodes "a" and "c" are pre-assigned to the same resource. The source code ordering has "a" before "c" before "f". Initially, list scheduling in step 1 will schedule T1 to cstep 0, and then fails to schedule T2 because of resource contention at cstep 0, and because its ALAP cstep is 0 nce T1 is scheduled t 0. In step 2, T1 will be assigned a relaxed cstep of 1. In the next iteration, list scheduling first schedules T2 to cstep 0, then schedules T1 to cstep 1 to avoid resource contention, and finally schedules T3 t cstep 3.

We have two recourses when the above algorithm fails: First, we can continue to try other scheduling algorithms which may still find legal schedules. Second, we can insert precedence constraints t sequentialize pre-assigned operations by their source code ordering. Any unsatisfiable maximum timing constraints would then be detected as positive cycles in the graph G.

5.0 Applications for Behavioral Templates

This section highlights how we use behavioral templates to advantage. Fig. 3 shows the overall flow of our scheduling process.

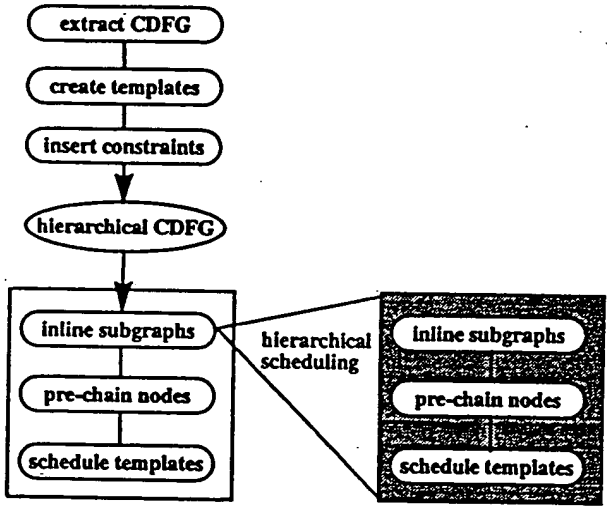


FIGURE 3. Overall flow for hierarchical scheduling

5.1 Inserting Timing Constraints

After extracting the CDFG and creating the initial templates, user-specified timing constraints are added to the CDFG. Minimum timing constraints are represented by precedence edges between nodes, but fixed timing constraints and maximum timing constraints are represented with the help of behavioral templates. Fixed timing constraints are when two or more operations must be scheduled in a fixed number of cycles apart. This is represented by adding one operation to the template of the other operation with the proper offset. For example, if n_j must start k csteps after n_i starts, and if n_i is in template $T = \{...(n_i, o_i)...\}$, then we add n_j to T at offset $o_i + k$ (Fig. 4(a)); if n_j must start k csteps after n_i *ends*, and n_i has a delay of d cycles, then we add n_j to T at offset $o_i + d - 1 + k$ (Fig. 4(b)).

However, if n_j must start k csteps after n_i ends, and n_i does not have a static delay (e.g., n_i is a subgraph), then we decompose the fixed constraint into a k-cycle minimum timing constraint from the end of n_i to the start of n_j, plus a k-cycle maximum timing constraint from the start of n_j to the end of n_i. This is shown in Fig. 4(c).

Note that in Fig. 4(c), we create a dummy *place holder* node, ph, and lock it in a template with n_j (k cycles apart). This template combines with the precedence edge of weight 0 from ph to the end of n_i

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX A
Sheet 4 of 7

THIS PAGE BLANK (USPTO)

to represent the maximum timing constraint from the start of n_j to the end of n_i. The precedence edge of weight k from the end of n_i to start of n_j, represents the minimum timing constraint from the end of n_i to the start of n_j.

In general, a maximum timing constraint of k cycles from a set of nodes, A, to the set of nodes, B, is represented by creating two place holders, ph1 and ph2, fixed k cycles apart in a template, and inserting a 0-weight precedence edge from ph1 to all nodes in A, and inserting a 0-weight precedence edge from all nodes in B to ph2. Fig. 9(a) contains an example of this where the dummy place holder nodes, t2 and t3, are used to lock a write operation to 0 cycle after the end of a loop.

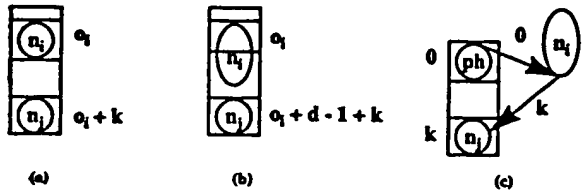


FIGURE 4. Timing constraints (a) n_j starts k cycles after n_i starts, (b) n_j starts k cycles after n_i ends and n_i has static delay d, (c) n_j starts k cycles after n_i ends and n_i's delay is not static.

5.2 Modeling Multi-cycle Operations

Behavioral templates also help model complex multi-cycle operations. When a single CDFG node is used to model a multi-cycle operation, it imposes some limitations due to CDFG semantics:

- Execution cannot start until ALL inputs are available.
- ALL inputs must be held stable throughout operation execution.
- ALL outputs are produced in the last cycle of execution.

This makes it difficult to model, for example, a 3-cycle RAM write operation where the address must be stable for the first two cycles, the data must be stable in the second cycle, and the write sequence finishes in the third cycle. To model such complex operations, we differentiate between combinational and sequential multi-cycle operations. A combinational operation has cycle synchronous inputs and outputs, so it is modeled by a single CDFG node. A sequential operation can have different cycle-by-cycle input/output connections and even resource requirements, and is modeled by several CDFG nodes that are locked into consecutive csteps by a behavioral template. Fig. 5 shows the single-node and multiple-nodes-in-a-template models for the above 3-cycle RAM write operation. Note that in our model (Fig. 5(b)), the address and data inputs are de-coupled in terms of when and for how long each input must be stable. This also de-couples multiple outputs (if any).

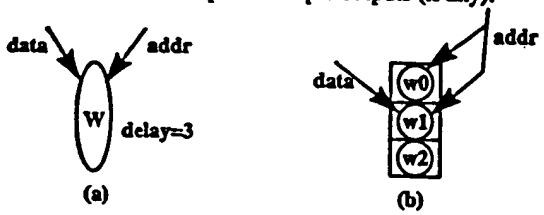


FIGURE 5. Models for a 3-cycle RAM write operation: (a) single node with delay = 3; (b) 3 nodes locked in a template.

This multiple-nodes-in-a-template model is even more powerful when resource requirements are added. For example, a pipelined operation uses different pipe-stage in different cycles, allowing overlapping pipelined operations to share the same hardware module as long as they do not have resource contention on any pipe-stage. If we view pipe-stages as internal resources and assign each pipe-stage a named "token", then we may label each node in the template model with the resource tokens it requires. As a node is scheduled to a cstep, we reserve its resource tokens for that cstep. The number of conflicting tokens (i.e., number of non-mutually-exclusive nodes that require the same token) in any cstep gives the number of pipelined modules needed in that cstep. Overlapping of pipelined operations can be scheduled on the same module because successive nodes in the template model require different tokens.

This removes assumptions about pipelined operations from the scheduling algorithms. We may now model operations on complicated pipelines, and template-based scheduling will properly schedule these operations on the pipelined modules. Fig. 6 shows examples of operations on pipelines with internal feedback, sequential inputs, and multiple outputs. We use "a[s1]" to denote an operation named "a" which requires the token "s1".

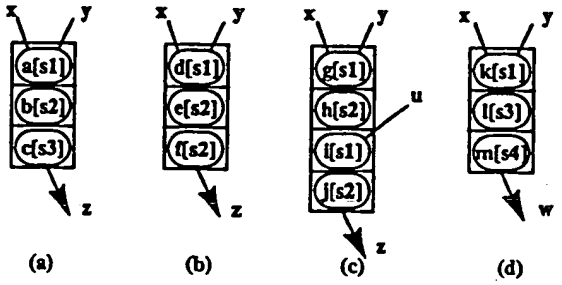


FIGURE 6. Template models for: (a) basic 3-stage pipelined operation, (b) 3-cycle pipelined operation with 2 stages and internal feedback, (c) 4-cycle pipelined operation with 2 stages and sequential inputs, (d) pipelined operation using a different internal path and output port.

Actually, resource tokens need not correspond to physical hardware resources, but may be considered a more general mechanism for specifying how different types of operations can overlap in time on the same module. Consider a 2-cycle RAM which has one read-port and one write-port, whose read/write cycles must be synchronized. Fig. 7 shows how we use resource tokens to specify this constraint to scheduling. If two such operations are pre-assigned to the same RAM, then resource contention on any of "s1", "s2", "s3" and "s4" implies an illegal schedule. The token "s3" prevents read operations for the same RAM to overlap; the token "s4" prevents write operations for the same RAM to overlap; and the tokens "s1" and "s2" prevent read and write operations for the same RAM from being scheduled exactly one cycle apart.

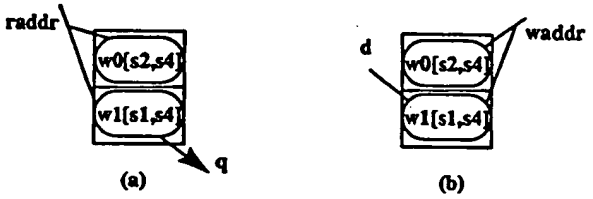


FIGURE 7. Template Models for RAM (a) 2-cycle read, and (b) 2-cycle write.

To handle multi-port RAM's, we allow a module to carry more than 1 copy of a given resource token. For example, to model a 4-port

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX A
Sheet 5 of 7

RAM where each port can be used for both read or write, we would define the RAM module t hav 4 "r/w" tokens, and model read and write peration on this RAM to require 1 "r/w" token each. This w uld allow scheduling to perform up t 4 simultaneous read or write operations on the same module.

5.3 Pre-Chaining

Just before scheduling, we selectively force operation chaining by locking operations in the same cycle using behavioral templates. This "pre-chaining" step reduces scheduling complexity at the expense of scheduling freedom. User-specified chaining directives are applied in this step. We also implement automatic pre-chaining for logic operations and bit-manipulation operations to save registers.

Logic operations include bit-wise AND, OR, NOT, EXOR operations, and bit-manipulation operations include bit-extract, bit-concatenate, constant bit/word generator operations. These operations are good candidates for pre-chaining because they have small propagation delays and they are not resource shared. Thus pre-chaining can be done on the basis of register costs alone. We implement a greedy algorithm for pre-chaining:

1.In a forward traversal of the data flow graph, pre-chain a logic/bit-manipulation operation with its predecessors if there are fewer output bits than input bits;

2. In a reverse traversal of the data flow graph, pre-chain a logic/bit-manipulation operation with its successors if there are fewer input bits than output bits.

3. Iterate until there are no more changes.

Fig. 8 shows examples of good pre-chaining configurations.

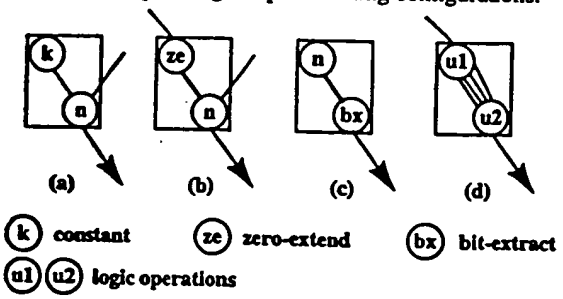


FIGURE 8. Pre-chaining examples: (a) constant with successor, (b) zero-extension with successor, (c) bit-extract with predecessor, (d) multi-input logic with predecessor or multi-output logic with successor

5.4 Hierarchical Scheduling

As shown in Fig. 3, our extracted CDFG is hierarchical, in which each level of the hierarchy corresponds to a loop or a subroutine. Hierarchical scheduling proceeds in a bottom-up traversal of the hierarchy. At each level, instead of representing subgraphs as super nodes, we inline each subgraph and use a behavioral template to interlock the inlined nodes according to the subgraph's schedule.

Inlining subgraphs allows certain boundary optimizations. First, unused subgraph outputs (and the operations that produce these outputs) can be deleted. This deleti n can recurs to unused subgraph inputs and then to operations that feed these inputs. Second, inlining subgraphs allows scheduling of neighboring nodes to take advantage f when individual subgraph inputs/outputs are actually required/produced, whereas representing subgraphs as super nodes would force scheduling to assume that all subgraph inputs/outputs are required/produced in the same cycles.

Another advantage of inlining subgraphs is that scheduling maintains accurate cycle-by-cycle resource costs. This allows, for example, to calculate resource costs of scheduling operations in the first and last cycles of a loop subgraph. (When an outside operation is scheduled in the first/last cycle of a loop it is performed when entering/exiting the loop).

In fact, hierarchical scheduling is used to implement sequential multi-cycle operations. In the initial CDFG, each sequential operation is a subroutine call to some library function, which is a pre-scheduled CDFG whose nodes are labelled with the required resource tokens. During inlining, each sequential operation is replaced by an inlined copy of its function's CDFG, and a template is created to lock these inlined nodes. This creates the multiple-nodes-in-a-template model for sequential operations.

The disadvantage of inlining subgraphs is that more nodes are scheduled instead of a small number of super nodes. This is balanced somewhat by the fact that inlined nodes are grouped by templates into a few scheduling units, so at least the scheduling solution space is not much bigger.

6.0 Results

Behavioral templates have been implemented in the Synopsys Behavioral Compiler™ product. Behavioral Compiler™ inputs a VHDL or Verilog behavioral description, performs scheduling, allocation, module selection, binding, and control optimization, and outputs a RTL design which is then optimized by RTL optimization [2], FSM optimization, and logic synthesis.

We will use "dft", a discrete fourier transform design, to illustrate behavioral templates. On reset, dft sequentially reads in the real and imaginary parts of the coefficients into arrays cmem and dmem. These arrays are mapped to the memory "CRAM" (cmem in the lower bank, dmem in the upper bank). It then enters the main processing loop. In each iteration, dft signals it is ready for processing, do a busy wait for the "start" signal, and then sequentially reads in the real and imaginary parts of the data points into arrays amem and bmem, which are also mapped to two halves of a memory, "DRAM". It then enters two nested FOR loops which compute the discrete fourier transform values and write them out. The memories are two-cycle RAM's whose read/write models are shown in Fig. 7. The multiply operations are done on a 2-stage pipelined multiply.

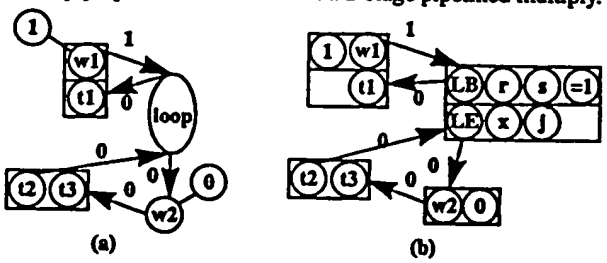


FIGURE 9. Handshaking for start signal: (a) original CDFG with timing constraints, (b) final CDFG scheduled

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX A
Sheet 6 of 7

THIS PAGE BLANK (USPTO)

Fig. 9 shows the CDFG fragment for the busy-wait on the "start" signal. Fig. 9(a) shows the initial CDFG containing the fixed constraints that the "ready" signal be asserted one cycle before the busy wait, and deasserted 0 cycle after the busy wait. Fig. 9(b) shows the same CDFG fragment that is finally scheduled. By this time, hierarchical scheduling has already scheduled the busy wait loop, and the loop body is inlined and locked in a template. Also, pre-chaining has locked the constants with the write operations.

However, the main scheduling problem is in the inner-most loop, which reads from memories the complex data *(a , jb)*, and coefficient *(c , jd)*, and computes *psum* += *(a*c - b*d)* and *ipsum* += *(a*d + b*c)*. Table 1 shows the scheduling and allocation results for the computation part of this loop. Note the pipelined RAM read's are chained with the pipelined multiply operations.

amem	bmem	cmem	dmem	p-mult		+/-
r0[s1]			r6[s1]			
r1[s2]	r2[s1]	r4[s1]	r7[s2]	x0[s1]		
	r3[s2]	r5[s2]		x1[s2]	x2[s1]	
				x4[s1]	x3[s2]	add1
				x5[s2]	x6[s1]	add2
					x7[s2]	sub1
						add3

TABLE 1. Scheduling/allocation results for dft's inner loop

FIGURE 10. Scheduling/allocation result for computation part of dft's inner-most loop

In Table 2 and 3, we present some design statistics. #line is number of VHDL/verilog lines in the source. #loop is number of loops with nesting levels in brackets. #node is number of CDFG nodes. #template is total number of templates scheduled. The ratio of nodes to templates are shown in brackets. #RAM is the number of on-chip memories used. #gate is gate count after logic synthesis (excluding RAM's). Note the difference between #node and #templates.

Table 2 presents several HLSW benchmarks, modified to use more realistic bit-widths. EWF is the fifth order elliptic wave filter example (20 csteps for the main loop, using 1 16x16 pipelined multiply, 2 32-bit adders, 10 32-bit registers and 1 16-bit register). KF is the Kalman filter modified to use 5 RAM's.

	#line	#loop	#node	#template	#RAM	#gate
EWF	128	2 (2)	103	78 (1.32)	0	9677
KF	207	10 (4)	261	144 (1.81)	5	7963
i8251	596	54 (3)	1238	625 (1.98)	0	5274
gcd	57	2 (2)	77	22 (3.50)	0	825

TABLE 2. Design statistics for several HLSW benchmarks

	#line	#loop	#node	#template	#RAM	#gate
dft	218	5(3)	151	65 (2.32)	2	3920
rgb filter	247	3 (2)	286	109 (2.62)	4	4316
idct	274	8 (3)	902	287 (3.14)	3	32677
viterbi	262	3 (2)	1797	296 (6.07)	0	3653
graphics ctrl	123	2 (2)	98	30 (3.27)	0	4071
high pass	389	5 (2)	310	153 (2.03)	11	8970

TABLE 3. Design statistics for several industrial examples

Table 3 lists several industrial examples. Compared to benchmark examples, these designs tend to:

- have more complicated reset sequences before the main processing loop,
- have more cycle-by-cycle timing constraints on IO operations,
- have more logic operations and bit-manipulation operations,
- use multiple RAM's or multi-port RAM's to improve RAM access bottlenecks,
- use pipelined operations to increase throughput.

7.0 Conclusion

In this paper, we have presented our work on scheduling using behavioral templates. The most important value of behavioral templates is that they enable simple solutions to the problems of (1) enforcing fixed and maximum timing constraints, (2) modeling complex sequential operations, (3) pre-chaining of logic and bit-manipulation operations, and (4) hierarchical scheduling. For this reason, behavioral templates have been instrumental in our productization of behavioral synthesis.

Future work will investigate adding structural templates to partition the design based on structural regularity.

8.0 Acknowledgment

We would like to acknowledge Russ Segal and Dennis Fogg, who designed and implemented library interface for our sequential operations. We would also like to acknowledge Pradeep Fernandes and Hazem Almusa for helping us collect the results reported in Tables 1, 2 and 3.

9.0 References

[1] R. Camposano, "Path-based scheduling for synthesis", IEEE transactions on Computer-Aided Design, vol. CAD-10, pp 85-93, Jan 1991.

[2] B. Gregory, D. MacMillen & D. Fogg, "ISIS: A System for Performance Driven Resource Sharing", in Proc. of 29th DAC, pp285-290, June 1992.

[3] D. Ku & G. De Micheli, "Relative Scheduling under Timing Constraints", in Proc. of 27th DAC, pp59-64, June, 1990.

[4] M.C. McFarland, A. C. Parker & R. Camposano, "The High-Level Synthesis of Digital Systems", in Proc. of IEEE, vol. 78, pp301-318, Feb. 1990.

[5] P. Michel, U. Lauther and P. Duzy, "The Synthesis Approach to Digital System Design", Kluwer Academic Publishers, 1992.

[6] S. Note, W. Geurts, F. Catthoor & H. De Man, "Cathedral III: Architecture-Driven High-Level Synthesis for High Throughput DSP Applications", in Proc. 28th DAC, pp597-602, June, 1991.

[7] M. Nourani & C. Papachristou, "Move Frame Scheduling and Mixed Scheduling-Allocation for the Automated Synthesis of Digital Systems", in Proc. 29th DAC, pp99-105, June, 1992.

[8] D. S. Rao & F. J. Kurdahi, "Partitioning by Regularity Extraction", in Proc. 29th DAC, pp235-238, June, 1992.

[9] D. S. Rao & F. J. Kurdahi, "System Modeling for High-Level Synthesis Using Regularity Extraction", D. S. Rao & F. J. Kurdahi, in Proc. Sixth International Workshop on High Level Synthesis, pp267-272, Nov 1992.

[10]K. Rose & C. T. Chang, "Cluster-Oriented Scheduling in Pipelined Data Path Synthesis", in Proc. ICCD, Oct. 1993.

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX A
Sheet 7 of 7

THIS PAGE BLANK (USPTO)

Appendix B

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street.
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX B
Sheet 1 of 7

THIS PAGE BLANK (USPTO)

Behavioral Synthesis Methodology for HDL-Based Specification and Validation

D. Knapp, T. Ly, D. MacMillen, R. Miller

Synopsys Inc.
700B E. Middlefield Rd
Mountain View, CA USA 94043

Abstract

This paper describes a HDL synthesis based design methodology that supports user adoption of behavioral-level synthesis into normal design practices. The use of these techniques increases understanding of the HDL descriptions before synthesis, and makes the comparison of pre- and post-synthesis design behavior through simulation much more direct. This increases user confidence that the specification does what the user wants, i.e. that the synthesized design matches the specification in the ways that are important to the user. At the same time, the methodology gives the user a powerful set of tools to specify complex interface timing, while preserving a user's ability to delegate decision-making authority to software in those cases where the user does not wish to restrict the options available to the synthesis algorithms.

1.0 Overview

This paper describes a synthesis methodology that uses high-level synthesis (HLS) of behavioral hardware-description language (HDL) descriptions. HLS has the distinguishing characteristic that operations are automatically *scheduled*, i.e. assigned to states, as opposed to lower-level synthesis, in which operations are assigned to states by the user [1, 2, 3]. For example, in an HDL description of a square root function, an operand x would be loaded, a series of operations would f ll w, and a single result r would be returned. The read x and the write r might be fixed to particular states or times by a communication protocol, but the internal operations that compute the square root would be automatically scheduled.

A prospective user of HLS will then ask a number of questions. These will likely include the following:

- H w can I constrain I/O operations to fall into particular cycles, or range of cycles, to meet existing protocols?
- How can I constrain I/O operations to have particular timing relationships? For example, how can I constrain a data ready strobe to be synchronous with data on data ports?
- How can I be confident that my interface timing specification really works with the surrounding hardware?
- H w can I give the scheduling software optimization opportunities when my timing specification is not rigid? For example, I might not care exactly when data was transferred, as long as a corresponding strobe remains synchronized with the data. Thus the strobe and data should be locked together, but the locked strobe/data pair of operations could move.
- How can I be confident that the synthesized hardware will really do what I want it to:
  1. In the sense that it computes the right result,
  2. In the sense that scheduling of I/O operations does not 'break' its I/O protocols.

These questions can be reformulated as requirements on the HDL description methodology to be used in conjunction with HLS:

- The original HDL description should be simulatable.
- There should be a mode wherein the cycle by cycle I/O timing of the original HDL description is preserved exactly; i.e., no I/O timing difference will be allowed between the pre-and post-synthesis descriptions. This will allow direct comparison, on a cycle by cycle basis, of the pre- and post-synthesis designs; it will also allow the user to meet the most rigid cycle-based timing protocols.
- There should be a mode wherein timing relationships between I/O signals can be simply and easily preserved across synthesis, but where 'stretching' (cycle level delay insertion) is permitted, so that the user does not have to specify exactly how many cycles a computation will take. This mode should allow manual constraints. Such a mode allows comparison of pre- and post-synthesis I/O timing between "similar points" of the pre- and post-synthesis waveforms.
- There should be a mode in which the user explicitly specifies all timing constraints without reference to the simulation behavior of the HDL; the only timing constraints inferred from the HDL description are ordering constraints among I/O operations sharing a port. This mode gives the greatest flexibility, both for optimization and for specification of complex timing relationships; it is also the most difficult to use.

We call these three modes the *cycle-fixed* IO *scheduling mode*, the *superstate-fixed IO scheduling mode*, and the *free-floating IO scheduling mode* respectively. Each has consequences for the style of HDL description and validation methodology. These modes give the user a wide range of choices in specifying I/O timing, with a corresponding range of ways in which validation of the specification and comparison of the implementation with the specification can be performed.

1.1 Structure of this paper

The balance of this paper is structured as follows. In Section 1.2, related work in this field is discussed. Following that, in Section 2, some mode-independent considerations and assumptions are described. In Section 3, the cycle-fixed mode is described in detail. Then in Section 4, the superstate-fixed mode is described. In Section 5, the free-floating mode is described. In Section 6, experience with the current software is described; finally, in Section 7 the paper is summarized and conclusions are drawn.

1.2 Related Work

High-l vel synthesis has been well described in the literature; see, for example, Camposano[1], Gajski[2], Maerz[3]. These tutorial papers describe the basics f HLS systems. CALLAS [4] describes work in the area of maintaining simulated behavior that is exactly the same pre- and post-synthesis; this idea is reflected

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USPTO)

in the cycle-fixed mode described here. The superstate-fixed mode is related the High Level State machine f of [5], and to the behavioral finit state machines (BFSM's) of [6]. Our approach of validation through simulation is typical f current industry practice; it complements, but cannot completely replace, more formal methods [7].

2.0 Basic assumptions

The circuit to be synthesized by HLS consists of a collection of always blocks (VHDL processes); each always block will be mapped to hardware consisting of a datapath and a control FSM. Each will be synthesized separately.

Control over timing makes use of clocking statements in the source HDL. In Verilog, this can be done by use of @(posedge clock) or @(negedge clock) statements[1]. These are used to separate I/O events that are to happen in different clock cycles. Event triggers using other signals are specifically disallowed, with the exception f asynchronous reset and a special gating methodology described in Section 2.2, used for synchronizing I/O.

2.1 Reset

In order to handle resets in an intuitively appealing way, we call attention to the always block (VHDL process) that will be scheduled. In our methodology this block contains a single all-encompassing, nonterminating loop, here called *reset_loop*.

```
always begin: bl
    begin: reset_loop
        // reset sequence behaviors
        forever begin
            // normal mode behaviors
        end
    end
end
```

Inside *reset_loop* is a *reset sequence*; this consists of all behaviors associated with reset. For example, in a microprocessor the reset sequence would clear the program counter, disable interrupts, and initialize the stack pointer. The reset sequence may contain many clock cycles, e.g. to initialize a RAM. Following the reset behavior is the 'normal mode' loop, which does not terminate either; this loop contains behaviors that are executed until the next reset occurs. In a microprocessor, for example, the normal mode loop would be the fetch / execute cycle.

In order to simulate the effect of synchronous resets correctly in the source HDL description, the user must insert a statement of the form[2]

```
if (reset == 1'b1) disable reset_loop;
```

after every @posedge statement. This disable has the effect of restarting the block (process) following a clock edge upon which reset is found to be true. Simulation of synchronous resets can be matched both pre- and post-synthesis.

Another capability can also be provided in which the user declares a reset pin to the synthesis software, which then synthesizes the reset; but because the reset behavior is not encoded in the HDL, resets cannot be simulated correctly before synthesis using this technique.

Scheduling cannot handle exits triggered by a reset in the same way as ther exits, because there may be read-before-write accesses in the HDL. Consider the following: In this situation, the assignments

```
begin: reset_loop
    outport <= x; // x is read bef re write!
    begin: main_loop
        x = v1;
        @(posedge clock);
        if (reset == 1'b1) disable reset_loop;
        x = v2;
    end
end
```

of *x* cannot be rescheduled, because this would change the observable behavior of the circuit immediately following a reset pulse. If, for example, the second write to *x* was rescheduled before the clock edge, then the output immediately following a reset pulse would be *v2* in the scheduled design; but it would be *v1* in the original description. So if we are to allow read before write in the HDL, we must either relax the requirement that all behaviors must be identical, or we must forbid movement of such side effects across clock boundaries. Side effects on variables that are always written before they are read are not affected.

2.2 Registered outputs

VHDL signals and Verilog reg variables behave like register or latch outputs. That is, they hold their values once set. For implementation reasons, we chose to register all outputs of HLS synthesized designs; thus a nonblocking (signal) assignment becomes a register write. This has the consequence that responses to external events cannot happen until the cycle after the external event, as shown in Fig. 1.

Figure 1 shows the behavior of a synthesized circuit where the HDL input is of the general form

```
if (Ready == 1'b1) then Data <= foo;
@(posedge clock);
```

This timing corresponds to both input and output. Notice that this

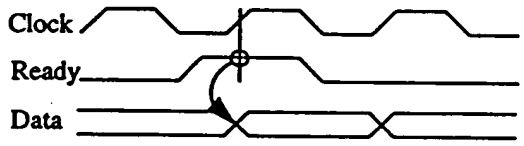


Fig. 1. Response to an external event.

timing diagram implies that the control FSM for the synthesized data path is a Mealy machine; and that the overall synthesized design is a Moore machine.

Here is an example combining an asynchronous reset and a compact busy wait on a data strobe.

```
while (strobe != 1) begin
    @(posedge clock or posedge reset);
    if (reset == 1'b1) disable reset_loop;
end
```

3.0 Cycle-fixed mode

High-level synthesis in *cycle-fixed* mode can be described by the following statement:

- Cycle-by-cycle I/O timing is identical between the pre-and post-synthesis designs.

This means that validation by simulation is straightforward: a user need merely simulate the pre- and post-synthesis designs side by side, and check for differences in the outputs. Alternatively, the synthesized design can be inserted into the original test bench without modifying the test bench. The only differences that are visible involve combinational delays in the form of setup and hold times; for example, a delta-delay setup time would become a real setup

1. In VHDL "wait until clock'event and clock = '1';" gives us a rising-edge clock.

2. In VHDL this would be "when reset = '1' exit reset_loop".

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USPTO)

time, and a registered output pin will not transiti n exactly on the clock edge, as it would in the pre-synthesis simulation[1]. This is shown in Fig. 2.

Notice that this mode only constrains the I/O operations of the design. That is, the reads and nonblocking (signal) writes of the HDL are tied to particular cycles. But this still leaves optimization pportunities for the scheduling algorithm: other operations (e.g. additions, memory operations, and register reads and writes) can be shifted in time, as long as they consume data after it has been read in, and produce data in time to write it out. The I/O operations provide a series of 'stakes in the ground' that define time frames within which all other operations are free to move.

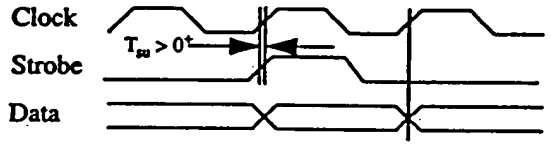


Fig. 2a. Simulation of specified design (pre-synthesis)

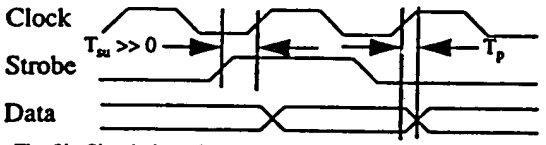


Fig. 2b. Simulation of synthesized design (post-synthesis)

Fig. 2. Comparison of simulation in cycle-fixed mode.

The main advantage of cycle-fixed mode is that the user can synthesize exactly the same timing diagram that the original HDL specification shows in simulation; thus, if the simulated HDL specification works in a particular context, then the synthesized design will also w rk, assuming only that setup, hold, and propagation delays, etc. as shown in Fig. 1b meet the clock cycle time.

A further advantage of cycle-fixed mode is that simulation of a zero-gate-delay model of the synthesized design will match the original specification exactly; hence a simple file difference program can be used to compare pre- and post-synthesis designs. This is expected to have a profound effect on user acceptance of HLS as a viable tool in the design cycle: users are able to simply and efficiently check the equivalence of designs before and after synthesis.

There are a number of methodological and implementation considerations that affect the way we can write and implement cycle-fixed mode. These will now be described.

3.1 Numbers of clock edges

One consequence of the commitment to maintain exact I/O equivalence in cycle-fixed mode is that numbers of clock edges cannot be varied inside the scope of loops and conditionals. To do so would distort the I/O timing of the design.

3.2 Lo p boundaries

Every loop f an always block must contain at least one clock edge statement. The only exception to this is loops with constant iteration bounds, which can be unrolled during synthesis.

A loop can be thought of as a subgraph of a finite-state machine (FSM) which forms a cycle. The synthesized design will enter this cycle when the loop is executed, and leave it when the loop is exited. Such a loop is shown in Fig. 3.

```
o1 <= v1;
while (c) begin: loop
    o2 <= v2;
    @(posedge clock);
end
o3 <= v3;
@(posedge clock);
```

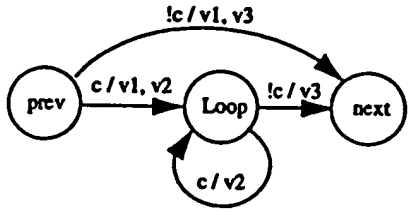


Fig. 3. Loop and corresponding state graph

The loop of Fig. 3 corresponds to the state labeled 'Loop'. During each pass of the loop, the value of *v2* will be written to the output port *o*.

The main consequence of matching this behavior is the splitting f the conditional test *c*. Notice that it was necessary, in order t capture the timing of the original, to have a state transition that bypassed the loop altogether if *c* was false when it was first tested. This means that the test must be performed in two places: once in state *prev*, and once in state *loop*. In general, it is necessary to unroll the first state of the first pass through a while loop in order t capture this behavior correctly.

If we wish to avoid unrolling the first pass, then it is necessary to rewrite the loop so that (1) there is a clock edge on all paths between the writes of *o1* and *o3*, and (2) there is a clock edge between the conditional test and any succeeding I/O, as shown in Fig. 4.

```
o1 <= v1;
while (c) begin: loop
    @(posedge clock);
    o2 <= v2;
end
@(posedge clock);
o3 <= v3;
```

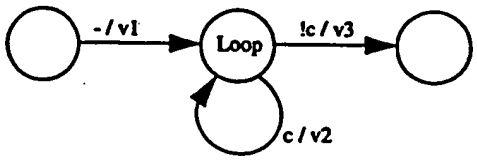


Fig. 4. Loop that does not need partial unrolling.

3.3 Conditional multicycle operations

A *multicycle* operation is one that has a longer combinational delay than the clock cycle. This imposes special constraints n synthesis in cycle-fixed mode, because it is necessary to stabilize all data and control inputs to the hardware block that implements the multicycle operation. This includes all the control inputs of all multiplexers

1. In zero-delay simulation one should ensure that data transitions occur slightly after clock transitions; failing to do this is the most common source of simulation mismatches. The problem comes about because of varying numbers of simulation-cycle delays in th clock and data wires of the circuit: the clock can arrive 'after' the data by an infinitesimal (zero-time) amount. This causes something analogous to a setup-time violation.

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USPTO)

that drive multicycle operations; clearly we cannot afford glitches n these paths.
But inserting these registers means that we need to know what t strobe into the registers ne cycle before the multicycle peration is t begin. Thus we need to add xtra time, under some circumstances, so that the stabilizing registers can be properly loaded. This is illustrated in Fig. 5; we assume

```
@(posedge clock);
if (input_signal == 1'b1) begin
        x = input_read_1;
        y = input_read_2;
        tmp = x + y; // 2 cycle addition
        @(posedge clock); // strobe stab regs
        @(posedge clock); // 1st cycle of add
        @(posedge clock); // 2nd cycle of add
        out <= tmp;
end
@(posedge clock);
```

Fig. 5. HDL description for a multicycle addition.

Notice that we needed three clock cycles to do this properly: one to get the condition and strobe the stabilizing registers, and two to perform the multicycle addition. Notice also that such delays can often be hidden, where the multicycle operations are not constrained by I/O; but that in this case there is no opportunity to hide the additional delay associated with stabilizing the inputs.

3.4 Loop pipelining in cycle-fixed mode

Loop pipelining is a technique whereby a loop can be made to act like a pipeline. Thus the loop has a relatively long latency, i.e. the time from a data input to the corresponding data output; and a sh rter initiation interval, which is the rate at which data can be delivered to and read out from the loop. In cycle-fixed mode, and with some extra constraints in the other modes, a simple way to imply loop pipelining while maintaining timing equivalence is to use a delayed assignment (in VHDL, a transport delay) on the output statement. Suppose, for example, we have a loop whose latency is ten cycles, but whose initiation interval is two cycles; we can put an output write after the second clock edge statement, with a delay of eight cycles. This will simulate the same way both bef re and after synthesis.

```
while (condition) begin
    @(posedge clock); // 10 ns clock
    @(posedge clock);
    out <= #80 value; // delayed by 8 cycles
end
```

4.0 Superstate-fixed Mode

The *superstate-fixed* I/O mode is used where the I/O should inherit its general structure from the HDL, but where there is some freedom to shift I/O operations in time. Consider, for example, the two-wire handshaking protocol shown in Fig. 6.
The two-wire protocol is insensitive to the time between transiti ns; this makes it ideal for many applications. In a case like this, the only things we really need to assure in order to have correct timing are that (1) the signal transitions occur in the right order, and (2) that the transitions of *Strobe* and *Data* maintain a lockstep relationship. Beyond that, the user might not care very much how many clock cycles were inserted by scheduling; other design optimization criteria (such as the number f gates to compute the data value) might dictate more or fewer clock cycles for this transaction. The cycle-fixed mode is unsuitable f r this kind f loosened specification of timing: the user could be forced to edit the code many times, with varying numbers of clock edge statements each time, looking for the best implementation.

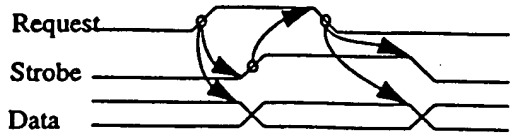


Fig. 6. Two-wire handshaking protocol.

The *superstate-fixed I/O scheduling mode* can be expressed by the following statements:

- Adjacent pairs of clock edge statements in the HDL form the boundaries of superstates.
- All I/O operations in a superstate remain in that superstate.
- A superstate may be expanded by the scheduler, which can add clock cycles to lengthen a superstate.
- All I/O writes in a superstate will always take place in the last clock cycle of the superstate.
- I/O reads may float within a superstate.

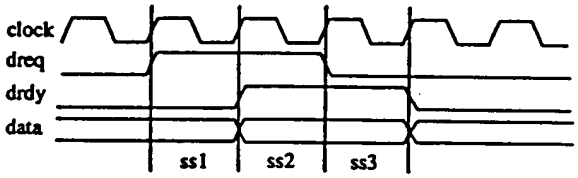


Fig. 7a. Simulation before superstate-fixed scheduling.

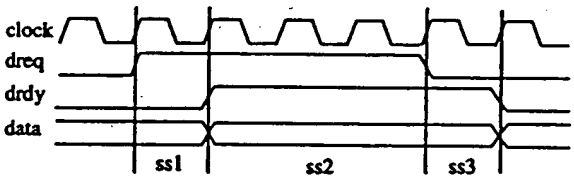


Fig. 7b. Simulation after superstate-fixed scheduling.

These rules, taken together, mean that an HDL scheduled in superstate mode will show the same signal transitions and ordering as the riginal HDL; but that the original timing may potentially be 'stretched' by the addition of new clock edges. This is illustrated in Fig. 7, where the original HDL simulation of an I/O transfer taking three cycles has become five cycles long by the addition of two extra cycles to the second superstate.

4.1 Protocols in superstate mode

One of the major advantages of superstate mode is that handshaking I/O protocols are not distorted by the addition of clock cycles to superstates. This has two beneficial conseqences: first, comparison of simulated pre- and post-synthesis designs is straightforward; and second, protocols that are insensitive to increased numbers of clock cycles will not be 'broken' by superstate scheduling. Hence if a design consists of many processes, each of which is to be scheduled, the use of handshaking communication in conjunction with superstate mode scheduling will ensure that the design will continue to work after synthesis. The same considerations apply to the simulation test bench as well: the test bench must communicate with the synthesized design(s) via handshaking protocols; otherwise it may have to be modified to communicate successfully with the synthesized design. This happens because the read and write operations occur at different times pre- and post-synthesis; the test bench must be able to tolerate this, or the user will have to retime the test bench.
Protocols that do not involve explicit requests and acknowledges can still be used; but care must be taken with data to be read in by the syn-

Atty. Docket No. 4000/10
Inventor: Tai A. Ly et al.
Title: METHODS FOR ATUOMATICALLY PIPELINING LOOPS

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

APPENDIX B
Sheet 5 of 7

THIS PAGE BLANK (USPTO)

thesized process. In particular, recall that read operations may move freely within their superstate. This means that data being presented to the synthesized circuit must be either valid during the entire superstate in which it is read, or else retimed after scheduling. This will ensure that the read operation always gets the correct data.

4.2 Constraints in superstate mode

The reason a designer would use superstate mode instead of cycle-fixed mode is that some part of the schedule does not have a fixed timing bound, and the user does not want to imply such a bound by using cycle-fixed I/O. However, the user may have a non-handshaking protocol, or a protocol that streams data once synchronization has been established by the protocol. In such cases the parts of the schedule that perform synchronization may need to be handled as if the scheduler was in cycle-fixed mode; while the other parts of the design can be allowed more freedom. For example, consider the fragment

```
while (ready == 1'b0) begin: handshaking_loop
    @(posedge clock);
end
@(posedge clock);
a1 = in_port; // label read_1
@(posedge clock);
a2 = in_port; // label read_2
@(posedge clock);
out_port <= long_involved_function(a1, a2);
out_ready <= 1'b1; // label done
@(posedge clock);
```

Here the external logic provides the data for *read_1* and *read_2* in the two cycles after the signal *ready* goes true; the synthesized system must pick it up then, or the protocol will be broken. Furthermore, insertion of extra cycles in the loop *handshaking_loop* will cause the interface to behave unpredictably. Thus cycle-fixed mode would seem to be indicated. However, suppose that there is no need for the output to show up until 20 cycles after the input has been delivered; the designer will thus want to allow the scheduler authority to add cycles to the last superstate, and rely on a test of the *out_ready* pin to synchronize the data on *out_port*. Thus stretching can be allowed in the last superstate, but not in the first three.

This can be done by means of explicit point-to-point scheduling constraints; that is, constraints that tie two labeled operations together in a particular timing relationship. A constraint set that would serve the purpose is

1. The time from the beginning of *handshaking_loop* to its end should be exactly one cycle.
2. The time from the end of *handshaking_loop* to the beginning of *read_2* should be exactly one cycle.
3. The time from the end of *handshaking_loop* to the data ready strobe *done* is no greater than 21 cycles.

Notice that these constraints are not part of the HDL; but they are a necessary part of the methodology. They can be implemented as pseudo-comments, as attributes, or as directives in a separate scheduler command file. Notice also that they can be applied to non-I/O operations as well, in all three modes, to give the user a little extra control over the scheduling process.

4.3 Superstate HDL methodology

Superstate mode defines superstates as containing the I/O operations that fall between adjacent pairs of clock edge statements. This definition has the consequence that sometimes an HDL prepared for superstate mode needs clock edge statements that are not needed in cycle-fixed mode. For example, the text of Fig. 3 is ambiguous when the HDL is considered as input for superstate mode. This comes about because two writes are separated by a conditional @posedge. If the loop condition is true, then the writes should be in different superstates; if it is false, then they should be in the same superstate. Clearly there is no unique static assignment of I/O operations to superstates in this situation.

Furthermore, there is an implicit ordering of operations conferred by the sequencing of the HDL text; this ordering cannot be allowed to come into conflict with the ordering conferred by the migration of reads into any cycle of their superstate and writes into the last cycle of their superstate.

The HDL methodology rules that prevent ambiguities and contradictions in superstate mode are:

1. A superstate that contains a loop continue is called a continuing superstate. Implicitly, the last superstate of a loop is also a continuing superstate. A continuing superstate and the first superstate of the loop are really the same superstate; there is no clock statement on the execution path going from one to the other. If a continuing superstate contains a write, then the first state of the loop cannot contain any I/O, because a write belonging to the continuing superstate would be migrated to the end of the first loop superstate: this would result in a violation of the HDL's ordering constraints.
2. A superstate that contains a loop beginning cannot include both an I/O write before the loop beginning and any I/O operation inside the loop. For example,

```
@(posedge clock);
out_port <= write1_data;
while (cond) begin
    read1_data = in_port; // Illegal!
    @(posedge clock);
    ...
end
```

   the write in this fragment conflicts with the read in the beginning of the loop; they are in the same superstate.
3. A write cannot precede a while loop that is succeeded by any I/O operation, unless there is a clock edge statement between either the write and the loop begin, or between the loop end and the second I/O operation.
4. A loop having a superstate in which both a loop exit[1] and an I/O write are located must have a clock edge statement between the loop end and the next I/O operation.
5. A conditional clock edge (e.g. an @edge on one branch of a conditional) cannot be used to separate a write from another I/O operation. This fragment is illegal for that reason.

```
out_port <= v1;
if (cond) @(posedge clock);
v2 = in_port;
```

5.0 Free-floating I/O mode

It will sometimes be the case that a user will need to convey more freedom to the scheduler than is allowed by the superstate I/O mode. For example, the user may wish to allow two unrelated writes to be permuted. Consider the fragment of Fig. 8.

In this situation, the user might not care whether the first or the second function happens first; indeed, they could be interleaved and the user might not care. But neither superstate nor cycle-fixed mode will permit permutation of I/O operations and waits; so a more powerful mode is needed.

---

1. Other than a reset exit. Resets can be ignored after a preprocessing step in which they are detected and global reset behavior is enacted, as explained in Section 2.

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USPTO)

The *free-floating* mode is characterized by implicit constraints on

```
a1 = in_port1;
a2 = in_port2;
@(posedge clock);
out_port_1 <= long_function_1 ( a1, a2 );
@(posedge clock);
b1 = in_port3;
b2 = in_port4;
@(posedge clock);
out_port_2 <= long_function_2 ( b1, b2 );
```

Fig. 8. Writes to out_port_1 and out_port_2 may be permuted.

single I/O ports and explicit user constraints.

Implicit I/O port constraints are derived directly from the HDL text and are imposed on the sets of reads and writes that occur on a single port. These are formed into partially ordered sets, one for each port, where the ordering is derived from a static execution trace analysis of the source HDL. The schedule constructed by synthesis can only transpose two members of one of these sets if there is no ordering relationship between them.

This, however, says nothing about ordering of reads and writes that occur on different ports, which must be explicitly constrained by the user, by means of the explicit two-point constraints described in Section 4.2.

For example, in our experience a common early mistake in free-floating mode is to expect a data strobe's timing to be fixed with respect to that of the data being strobed. This will not necessarily be the case if the user does not issue explicit constraints.

The downside of this mode is the number of explicit constraints that the user must construct. This can easily be comparable in numbers of lines to the HDL input itself. In addition, it is very easy to get such constraints wrong, or to forget a crucial constraint; hence the cycle-fixed and superstate modes are simpler and less error-prone to use.

6.0 Experience

Support for the methodologies discussed above has been built into a commercial product, the Synopsys Behavioral Compiler(TM). This product is currently in use at a number of sites. Of these, about half use Verilog as their input HDL; the rest use VHDL.

Experience to date indicates that the superstate mode is usually the most convenient from the standpoint of ease of specification of complex timing behaviors. The next most convenient is usually the cycle-fixed mode. The reason for this is that the power of the free-floating mode comes at the price of manually added constraints; while the cycle-fixed mode requires the user to add clock cycles to the source HDL when, e.g., the duration of a particular loop is to be changed.

From the standpoint of ease of validation of results, the cycle-fixed mode is usually a little more convenient than the superstate mode. This is because the handshaking protocols necessary to get the design talking to the test bench after superstate-mode scheduling must be designed and written in both the test bench and the specification; or alternatively the test bench timing must be modified to match the schedule of I/O of the post-synthesis design.

One area in which the free-floating mode seems to be more convenient than the others is in that of exploration. Here the user is more interested in getting a rough idea of the cost and speed of a design or algorithm, than in getting its interfaces exactly right. In this context, the ease of turning the design around and the high degree of freedom from methodological constraints makes it simpler to change the design and resynthesize to see what the overall results are. Then when the general outlines of th algorithms, representations, etc. are clear the user can begin to worry about the detailed I/O timing.

The overall effort of getting I/O interfaces right using these three modes is usually less than the effort spent in getting the best possible quality of results. Even with behavioral synthesis, HDL writing styles still can have a large impact on the quality of the synthesized circuit. Examples that can affect synthesis quality are: loop ordering, assignment of variables and arrays to memories, choice f loop pipeline initiation intervals and latencies, pipelined components, embedding combinational logic in reusable function blocks, the tradeoff between multicycle operations and fast clock rates, and the partitioning of the design into datapath/controller subunits (i.e. always blocks; in VHDL, processes). All are potentially of great importance to the quality of results, and all represent true engineering decisions that must be carefully considered if a really good design is to be achieved.

7.0 Conclusion

We have presented HDL methodologies for the synthesis of various kinds of I/O timing and protocols, and for simulation-based validation of the synthesized design against the original specification. Three modes of scheduling I/O operations have been presented:

1. Cycle-fixed, in which the design has exactly the same cycle-level I/O timing before and after synthesis;
2. Superstate-fixed, in which I/O operations are grouped by pairs of @posedge statements; post-synthesis timing behavior is a (potentially) stretched version of the pre-synthesis timing; and
3. Free-floating, in which the only constraints on I/O scheduling are either between operations sharing a port or supplied by the user.

Some of the implications of the scheduling modes were described. In the cycle-fixed and superstate modes, these involve the placement of clock edge statements, loop boundaries, conditionals, and I/O operations; while in the free-floating mode there are no rules of this kind.

Experience with production software which implements these methodologies has been described, and conclusions based on that experience have been drawn.

8.0 References

1. R. Camposano, W. Wolf. Trends in High-Level Synthesis. Kluwer, 1991.
2. D. Gajski, N. Dutt, A. Wu, S. Lin. High-Level Synthesis: Introduction to Chip and System Design. Kluwer, 1992.
3. S. Maerz, High Level Synthesis. In *The Synthesis Approach to Digital System Design*, P. Michel, U. Lauther, P. Duzy, eds., Chapter 6. Kluwer, 1992.
4. A. Stoll and P. Duzy, High-Level Synthesis from VHDL with Exact Timing Constraints, *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pp. 188-193, IEEE, 1992.
5. R. A. Bergamaschi, A Kuehlmann, S-M. Wu, V. Venkataraman, D. Reischauer, and D Neumann, A Methodology for Production Use of High Level Synthesis, Workshop Proceedings, Sixth International Workshop on High Level Synthesis, (1992).
6. W. Wolf, S. Takach, C.-Y. Huang, R. Manno, E. Wu. The Princeton University Behavioral Synthesis System. *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pp. 182-187, IEEE, 1992.
7. K. L. McMillan, Fitting Formal Methods into the Design Cycle, *Proceedings of the 31st ACM/IEEE Design Automation Conference*, pp. 314-319, IEEE, 1994.

Jonathan T. Kaplan, Esq.
Registration No. 38,935
Brown Raysman Millstein Felder & Steiner LLP
Attorney for Applicants
120 West Forty-Fifth Street
New York, New York 10036
Phone: (212) 944-1515 Fax: (212) 840-2429

THIS PAGE BLANK (USPTO)

EXHIBIT 3

THIS PAGE BLANK (USPTO)

THIS PAGE BLANK (USPTO)

EXHIBIT 3

TO

DECLARATION OF JONATHAN T. KAPLAN AND STATEMENT OF FACTS IN SUPPORT OF FILING ON BEHALF OF NON-SIGNING INVENTOR

Pursuant to 37 CFR 1.47

THIS PAGE BLANK (USPTO)

August 29, 2000
San Jose, California

Jonathan T. Kaplan, Esq.
Brown, Raysman, Millstein,
Felder & Steiner LLP
120 W Forty-Fifth St
New York, NY 10036

Dear Mr. Kaplan:

This responds to your letter of July 31, 2000, your file reference number 4000/10, in which you requested my signature on a reissue of US patent number 5764951.

I must inform you that I do not believe that the extension is novel; and further, I do not believe that it is what we invented. I must therefore decline tc sign the application.

Sincerely,

David W. Knapp
CTO
Get2Chip.com, Inc.

THIS PAGE BLANK (USPTO)